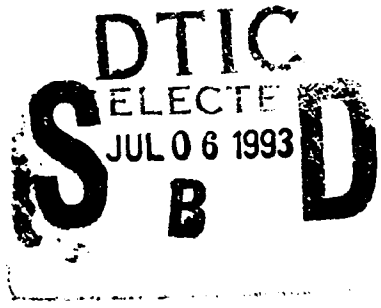
(1)

ACM SIGPLAN
Workshop on ML and its Applications

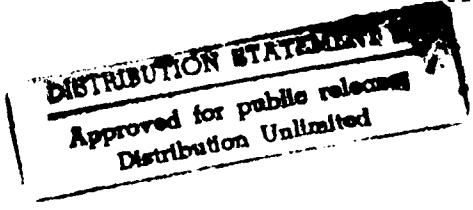
Peter Lee, *Editor*

June 1992
CMU-CS-93-105



School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

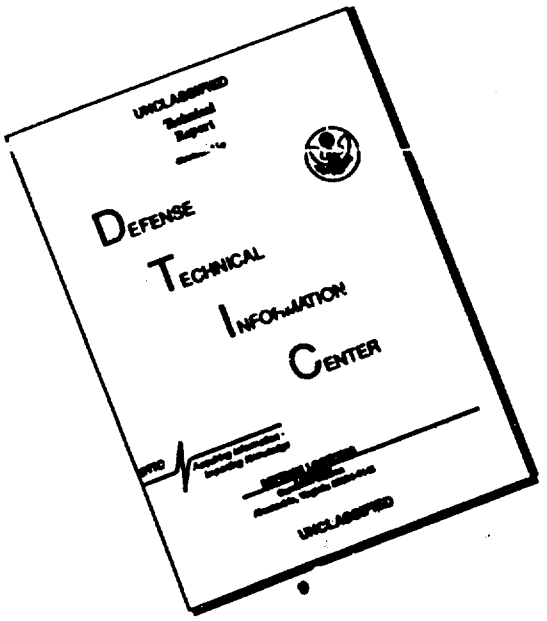
The Workshop on ML and its Applications was held June 20-21, 1992 in San Francisco, CA, in conjunction with the ACM SIGPLAN '92 Conferences and Workshops.



Abstract

The Workshop on ML and its Applications was held June 20-21, 1992 in San Francisco, CA, in conjunction with the ACM SIGPLAN '92 Conferences and Workshops. The ML programming language has been an important tool and framework for research in language design and implementation. As the language and its implementations have matured, the range of applications has increased greatly. New applications, combined with new ideas in design and implementation, have stimulated a large number of significant activities in research and software development. This workshop, like the previous workshops in Princeton, Edinburgh, and Pittsburgh, provided a forum for theses activities, with a special emphasis on applications of the language. A total of 21 abstracts were selected for presentation and discussion at the workshop: 12 were chosen for conference-style presentations and 9 for presentation in a poster session.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Keywords: Programming languages, ML

ACM SIGPLAN Workshop on ML and its Applications

San Francisco, California

June 20–21, 1992

Workshop Committee

General Chair

David MacQueen	AT&T Bell Laboratories

Program Chair

Peter Lee	Carnegie Mellon University

Program Committee

Simon Finn	Abstract Hardware, Ltd.
Emden Gansner	AT&T Bell Laboratories
Robert Harper	Carnegie Mellon University
Peter Lee	Carnegie Mellon University
Michel Mauny	INRIA
John Mitchell	Stanford University
Mads Tofte	University of Copenhagen

Foreword

The Workshop on ML and its Applications was held on June 20 and 21, 1992, in San Francisco, California, in conjunction with the ACM SIGPLAN '92 Conferences and Workshops.

The ML programming language has been an important tool and framework for research in language design and implementation. As the language and its implementations have matured, the range of applications has increased greatly. New applications, combined with new ideas in design and implementation, have stimulated a large number of significant activities in research and software development. This workshop, like the previous workshops in Princeton, Edinburgh, and Pittsburgh, provided a forum for these activities, with a special emphasis on applications of the language.

A total of 21 abstracts were selected for presentation and discussion at the workshop. Of these, 12 were chosen for conference-style presentations and 9 for presentation in a poster session. Four-page abstracts were submitted, almost all by electronic mail. The abstracts were not formally refereed, but were reviewed by all members of the program committee. Technical excellence, originality, and relevance to the theme of "applications" were the main selection criteria. Most of the abstracts that focused primarily on a particular application of ML, as opposed to a language extension or implementation, were grouped into the poster session. Unfortunately, many very good abstracts were not selected, in part to keep a good workshop focus, and also to leave enough room in the schedule for informal discussions. A goal of the program committee was to encourage participation by as many people as possible, while still retaining a productive workshop atmosphere.

We would like to thank all of the authors who submitted abstracts, whether accepted or not. The large number of submissions is a testament to the importance and excitement surrounding the ML language.

ACM SIGPLAN Workshop on ML and its Applications

San Francisco, California
June 20–21, 1992

Saturday, June 20, 1992

Welcome: 9:00–9:15 a.m.

Session 1: 9:15–10:45 a.m. Design and Implementation of ML.
Chair: Michel Mauny (INRIA)
Abstract Value Constructors
William E. Aitken and John H. Reppy (Cornell University) *1*
Efficient Representation of Extensible Records
Didier Rémy (INRIA-Rocquencourt) *12*
A General and Practical Approach to Concrete Syntax Objects within ML
Mikael Petterson and Peter Fritzson (Linköping University, Sweden) *17*

Session 2: 11:15 a.m.–12:30 p.m. Code Generators for ML.
Chair: Robert Harper (Carnegie Mellon University)
An Optimizing ML to C Compiler
Régis Cridlig (Ecole Normale Supérieure and INRIA-Rocquencourt) *28*
An Efficient Way of Compiling ML to C
Emmanuel Chailloux (LIENS-LITP, France) *37*
Standard ML for MS-Windows 3.0
Yngvi S. Guttese (The Technical University of Denmark) *52*

Poster Session 3: 2:30–4:30 p.m. Applications of ML.
Chair: Peter Lee (Carnegie Mellon University)

Sunday, June 21, 1992

Session 4: 9:30–11:00 a.m. Type System Extensions.
Chair: John Mitchell (Stanford University)
Completely Bounded Quantification is Decidable
Dinesh Katiyar and Sriram Sankar (Stanford University) *68*
An Extension of ML with First-Class Abstract Types
Konstantin Läufer (New York University) and Martin Odersky (Yale University) *78*
Dynamic Typing in Polymorphic Languages
Martin Abadi (DEC SRC), Luca Cardelli (DEC SRC), Benjamin Pierce (University of Edinburgh), and Didier Rémy (INRIA Rocquencourt) *92*

Session 5: 11:30 a.m.–1:00 p.m. Applications of ML.
Chair: Emden Gansner (AT&T Bell Laboratories)
Extensions to Standard ML to Support Transactions
Jeannette M. Wing, Manuel Faehndrich, J. Gregory Morrisett, and Scott Nettles (Carnegie Mellon University) *104*
Programming Images in ML
Emmanuel Chailloux (LIENS-LITP, France) and Guy Cousineau (LIENS, France) .. *119*
Distributed Programming with Asynchronous Ordered Channels in Distributed ML
Robert Cooper and Clifford Krumvieda (Cornell University) *134*

Poster Session Summaries.
A Verification Environment for ML Programs
A. Cant and M. A. Ozols (Defence Science and Technology Organization, South Australia) *151*
Expressing Fault-Tolerant and Consistency-Preserving Programs in Distributed ML
Clifford D. Krumvieda (Cornell University) *157*
Implementing ML on the Fujitsu AP1000
Peter Bailey and Malcolm Newey (Australian National University) *163*
Verification of Concurrent Systems in SML
Paola Inverardi (I.E.I.-C.N.R. Pisa), Corrado Priami (University of Pisa), and Daniel Yankelevich (University of Pisa and HP Labs, Pisa Science Center) *169*
A File System in Standard ML
Drew Dean (Carnegie Mellon University) *175*
Implementing a Mixed Constructive Logic in Standard ML
James T. Sasaki (University of Maryland Baltimore County) and Ryan Stansifer (University of North Texas) *181*
Experiences with ML for Building an AI Planning Toolbox
Tom Gordon, Joachim Hertzberg, and Alexander Horz (GMD, AI Division) *187*
Attribute Grammars in ML
Sofoklis G. Efremidis (Cornell University), Khalid A. Mughal (University of Bergen, Norway), and John H. Reppy (AT&T Bell Laboratories) *194*
ML and Parsing—A Position Paper
Nick Haines (Carnegie Mellon University) *201*

Abstract Value Constructors

William E. Aitken*
Cornell University
aitken@cs.cornell.edu

John H. Reppy†
Cornell University‡
jhr@research.att.com

April 10, 1992

1 Introduction

Standard ML (SML) has been used to implement a wide variety of large systems, such as compilers, theorem provers and graphics libraries; even operating systems have been contemplated. While SML provides a high-level notation for programming large applications, there are some missing language features. One such feature is a general mechanism for assigning symbolic names to constant values. We present a simple extension of SML that corrects this deficiency in a way that fits naturally with the semantics of SML [MTH90, MT91]. Our proposal generalizes SML's datatype constructors: *constants* generalize nullary datatype constructors (like `nil`), and *templates* generalize non-nullary datatype constructors (like `::`). Constants are identifiers bound to fixed values, and templates are identifiers bound to structured values with labeled holes. Templates are useful because they allow abstract access to user defined types. In the remainder of the paper, we give examples of the utility of our mechanism, informally explain its semantics, discuss its implementation, describe the interaction between our mechanism and the module system, and discuss related work.

2 ML datatypes and patterns

The SML datatype mechanism provides a high-level mechanism for declaring new structured types. For example, the declaration

```
datatype 'a list = nil | :: of ('a * 'a list)
```

*This work supported, in part, by the ONR under contract N00014-88-K-0409.
†This work was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.
‡Current address: AT&T Bell Laboratories, Murray Hill, NJ.

defines the standard predefined polymorphic list type. This declaration defines two *data constructors*: `nil` is a nullary constructor representing the empty list, and `::` (which is an infix operator) is the list constructor. The expression `1::2::3::nil` evaluates to an integer list of three elements (SML provides the derived notation `[1, 2, 3]` for this construction). Datatypes can also be used to define enumerations; for example,

```
datatype direction_t = NORTH | SOUTH | EAST | WEST
```

The utility of datatypes is greatly enhanced by the use of *pattern matching* in function definitions to do case analysis and value decomposition. For example, the following function takes a list of strings and returns a list with commas inserted between adjacent elements:

```
fun commas [] = []
  | commas [s] = [s]
  | commas (s::r) = s :: "," :: (commas r)
```

This function consists of three rules (or equations). The left side of a rule is a pattern, the right side is an expression to be evaluated when the pattern is matched. The first rule in `commas` matches the empty list. The second matches the singleton list and binds the list's element to `s`. Theoretically, the third matches any list of one or more elements, but, since the order of equations defines their precedence, it in fact matches only lists of two or more elements; it binds `s` to the head of the list and `r` to the tail. Patterns in SML are *linear*, i.e., a variable may occur at most once in a pattern.

The data constructors defined by datatypes have a dual nature; they are used both to construct values (when they used in expressions), and to destruct values (when they are used in patterns). We use the term *value constructor* to refer to identifiers with this dual nature. If no confusion with type constructors is possible, we abbreviate this to constructors. Our extension of SML provides another mechanism through which identifiers with this dual nature may be defined. Since the values constructed using the data constructors of a datatype belong to the datatype's type, they are effectively distinct from any pre-existing values. Our mechanism allows the definition of *constants*, that is value constructors that refer to fixed pre-existing values, and the definition of *templates*, that is value constructors that refer to fixed, structured values with named holes. Templates can be thought of as the constructors for families of pre-existing values in the same way that non-nullary data constructors may be thought of as constructors for families of new values.

3 Constants

The use of symbolic names to refer to constant values is an essential tool in the writing of understandable, maintainable software. Every systems programming language provides some sort of support for symbolic constants: C has a powerful macro pre-processor, Modula-3 has constant declarations, and so forth. In SML, `val` bindings allow values to be given symbolic names, but these names cannot be matched against in patterns. This is a serious

limitation because pattern matching is the principal mechanism for case analysis and the decomposition of structured values. It is also possible, using the `datatype` declaration, to declare identifiers that may be matched against in patterns; however, these are not symbolic names for existing values, but rather names for entirely new values. While these two mechanisms are adequate for many applications, sometimes what is required is a mechanism that allows identifiers both to be bound to particular values and to be matched against in patterns.

For example, consider the implementation of an X Window System library. The X Window System uses a device independent representation of keyboard keys called *keysyms*. We need to provide users with a symbolic name for each of them. Clearly, users need to be able to pattern match against these names so that they can conveniently write programs that respond to different keystrokes differently. Thus `val` bindings are not suitable to provide these names. Using a datatype to represent keysyms is also unsuitable. The *wire representation* of a keysym, that is, the representation used by the X server, is a 29 bit integer. If the library uses a datatype to represent keysyms, it must also include a function to convert from wire representations to library representations, and another to convert library representations back into wire representations. It is essential that these functions be mutually inverse. Since there are literally thousands of keysyms defined — even *minimum* English language support requires 512 — these functions present a maintenance nightmare. Furthermore, the use of such huge functions adds significant run-time overhead to the marshaling and unmarshaling of messages. Using our mechanism, we can write definitions such as

```
datatype keysym_t = KEYSYM of int
const KS_a = KEYSYM 97
const KS_b = KEYSYM 98
```

Here the representation of a keysym is the constructor `KEYSYM` wrapped around the wire representation. (This is a standard idiom for creating new types isomorphic to existing ones. Compilers can represent `keysym_t` and `int` identically, and treat the constructor `KEYSYM` as a no-op.) The identifiers `KS_a` and `KS_b` are declared as constants that represent the characters 'a' and 'b'. They can be used both in patterns and in expressions. The identifier `KS_a` stands for the structured value `(KEYSYM 97)`. Note that this representation of keysyms has a further advantage over the datatype representation in that it allows keysyms from different keysets to be defined in different modules, enabling users to import only those keysets they actually need.

This use of our mechanism is reminiscent of the `#define` mechanism of the C language [KR88] (arguably, this is one area in which C provides a higher-level notation than SML). SML has a more general pattern matching facility than the C `switch` statement, and our symbolic constants reflect that. For example, a calendar program might include the following definitions

```
datatype date = DATE of {month : int, day : int}
const CHRISTMAS = DATE {month=12, day=25}
```

4 Templates

Templates are a natural generalization of symbolic constants to allow labeled holes[1]. They provide a mechanism to define a concise syntax for a collection of similarly structured values. A template is defined by a declaration of the form

```
const id trivpat = patexp
```

where *trivpat* is a pattern that involves only variables and record construction, and *patexp* is a pattern that contains only variables, record construction, special constants and other value constructors. Every *patexp* can be viewed both as a pattern and as an expression (they are the syntactic intersection of patterns and expressions). Every variable appearing in *trivpat* must also appear exactly once in *patexp*.

For example, the template mechanism allows us to define a template for the days of the month July using the declaration

```
const JULY(x) = DATE{month=7,day=x}
```

This would allow expressions like JULY(17) to be used to create date values for days in July. It would also allow code such as

```
case day
 of JULY(4) => "Independence Day"
  | _ => "not Independence Day"
```

in which JULY is used as a constructor in a pattern match.

A more substantial and more useful example of templates arises in systems that do term manipulation (such as the Nuprl proof development system, or code optimizers). For the sake of concreteness, we set our example in the context of a generalized λ-calculus. A *term* is either a *variable* or the application of an *operator* to a sequence of *bound terms*. For example, λ and **ap** are operators and $\lambda(x.a)$ is a term with x bound in the sub-term a, and $\mathbf{ap}(a;b)$ is term with sub-terms a and b (but no bound variables). One possible representation of this term language uses a different constructor for each operator.

```
datatype term
  = VAR of var
  | LAMBDA of (var * term)
  | AP of (term * term)
  | PLUS of (term * term)
  | ...
```

This representation has a two major deficiencies. First, functions like substitution that are independent of operators need to be written using many similar cases. Second, often

[1]The term *template* was suggested by Dave MacQueen.

(for example, in Nuprl), it is desirable to make the set of operators extensible, but, if this representation of terms is used, the datatype needs to be changed to extend the operator set, and this requires a complete recompilation of the program. Furthermore, extension of the operator set exacerbates the problem with functions like substitution — the addition of a new operator requires that a new case be added to each such function. The following representation defines a regular, extensible structure.

```
datatype operator = OP of string
datatype term
  = VAR of var
  | TERM of operator * ((var list * term) list)
```

Unfortunately, the syntax of expressions and patterns using this representation is quite ugly. For example, the pattern that matches β-redices (i.e., terms of the form $\mathbf{ap}(\lambda(x.s); t)$ is

```
TERM(OP "AP", [([], TERM(OP "LAMBDA", [([x], s)])), ([], t)])
```

compared to

```
AP(LAMBDA(x, s), t)
```

in the first representation. Moreover, the second representation scheme does not provide the syntactic checking given by the first representation. `TERM (OP "LAMBDA", [])` is a perfectly acceptable member of the type `term` even though terms formed with the λ operator should always have exactly one subterm in which exactly one variable is newly bound. Additionally, the use of strings to name operators adds the overhead of string comparison to pattern matching. All these problems are nicely solved using our mechanism. For example, with the following declarations

```
datatype operator = OP of int
datatype term = VAR of var | TERM of operator * ((var list * term) list)
cons: LAMBDA_OP = OP 0 and AP_OP = OP 1 and PLUS_OP = OP 2 and ...
cons: AP(p, q)     = TERM(AP_OP, [([], p), ([], q)])
  and LAMBDA(x, t) = TERM(LAMBDA_OP, [([x], t)])
  and PLUS(a, b)   = TERM(PLUS_OP, [([], a), ([], b)])
```

the pattern for β-redices is again `AP(LAMBDA(x, s), t)`. While the `term` type still includes many unintended values, disciplined use of the templates `AP`, `LAMBDA` and `PLUS` makes it impossible for users to produce these values accidentally. The required discipline can be enforced using the module facility. Thus, we get both the succinctness and syntactic checking of the first representation and the flexibility and regular structure of the second.

5 Constructors and the module system

The module system is an important feature of the SML language. Our proposal meshes elegantly with the module system, and the module system makes the mechanisms of our proposal even more powerful. Because the module system allows the programmer to limit the externally visible definitions of a structure, it is possible to limit the constructors available to users of the structure. For example, in the term example of the previous section, it might be desirable to limit users to the constructors `AP`, `LAMBDA` and `PLUS`, without giving them access to the lower level constructors such as `TERM`, and `LAMBDA_OP`. This would make it impossible for users to create junk terms.

Because data constructors are just a special kind of value constructor, it is possible to provide an interface to a structure in which they are made available as constructors without having to make their declaration as datatype constructors visible. This in turn allows the constructors of a datatype to be selectively exported. This is useful if the datatype declaration includes private constructors that are used to form intermediate values not valid in input or output.

Our proposal adds constructor specifications to the syntax of signatures. Nullary constructors are specified by the specification

```
const id : type
```

and unary constructors are specified by the specification

```
const id : type of type'
```

In this specification *type'* gives the type of the argument, and *type* gives the type of the constructed values. The syntactic distinction between unary and nullary constructors is required because, in patterns, unary constructors must always appear with arguments and nullary constructors may never appear with arguments. The legality of code such as

```
signature SIG =
  sig
    type unknown
    const K : unknown
  end

functor F (A : SIG) =
  struct
    fun f A.K = 17
      | f _ = 12
  end
```

depends on `K` being a nullary operator. Thus it is essential that structures such as

```
structure S = struct
  type unknown = int -> int * int
  const K x = (x, 12)
end
```

not be allowed to match SIG. We do this by explicitly associating arities with constructors rather than inferring them from type information as is done in the standard semantics. This in turn requires that the arity of constructors be available in signatures.

6 Informal Semantics

We have adapted the formal semantics of SML given in [MTH90, MT91] to handle value constructors and templates. Because of space limitations, we can only sketch the important issues here; the interested reader is referred to [AR] for the details.

Clearly, our proposal requires that a value constructor identifier can denote different values in different program contexts. For example, after the declaration

```
const MyNumber = 1
```

the constructor MyNumber must denote the integer 1, while after the declaration

```
const MyNumber = 2
```

it must denote the integer 2. To this end, an environment is used to associate constructor identifiers with their values. Moreover, the values associated with these identifiers must provide sufficient information to encode their behavior in patterns as well as in expressions.

The semantics of constants is straightforward. We use the environment to associate each constant identifier c with its value v. Nullary data constructors can be treated in this way: they are bound to themselves in the environment. When c is used in an expression, its value is v. Attempts to match a value w against c succeed if $w = v$ and fail otherwise.

The semantics of a template C is given using a pair of functions (C_π, C_ι): an injection and a projection. The injection is used to construct values in expressions, and the projection is used to perform the data destructuring associated with patterns. Non-nullary data constructors can be treated in this manner. Using ML-like notation, the functions corresponding to the data constructor TERM defined above are

$$\begin{aligned} \texttt{TERM}_\iota &= \texttt{fn x => TERM x} \\ \texttt{TERM}_\pi &= \texttt{fn (TERM x) => x | \_ => FAIL} \end{aligned}$$

where FAIL is a special value used to denote match failure. Similarly, the functions associated with the identifier LAMBDA by the template definition given above are

$$\begin{aligned} \texttt{LAMBDA}_\iota &= \texttt{fn (x, t) => TERM(LAMBDA\_OP, [([x], t)])} \\ \texttt{LAMBDA}_\pi &= \texttt{fn (TERM(LAMBDA\_OP, [([x], t)])) => (x, t) | \_ => FAIL} \end{aligned}$$

Note the appearance of LAMBDA_OP in these functions. Since the scope of LAMBDA_OP may differ from that of LAMBDA these functions must record the environment in which they were defined.

The most important point in the semantics of the module system is the appropriate definition of signature-structure matching. Informally a signature Σ matches a structure S if it is less specific: i.e., has fewer components, less polymorphism, or elides the constant nature of values. For example, the specification

```
val nil : int list
```

matches the standard list constructor `nil` (which has type `'a list`). The specification

```
const nil : int list
```

also matches the constructor `nil`. This can be used to export a limited view of a datatype. When a structure is constrained by a signature (e.g., when used as the argument to a functor), it is necessary to *thin* it by removing the unused components. Thinning also involves mapping constructors to values by discarding their projection functions. To facilitate the definition of structure thinning, our semantics uses separate environments to associate the injection and projection components of a constructor's value with the constructor rather than using a single environment and associating explicit injection-projection pairs with each constructor.

7 Extensions

In addition to datatype declarations, SML includes two other flavors of value constructors: exception constructors and the special constructor `ref`. Treating exception constructors as value constructors is entirely straightforward. Every time an exception declaration is evaluated, a new exception name is generated, and associated with the declared exception identifier in both the value and projection environments. Treating `ref` as a value constructor is more difficult. Such a treatment of `ref` would allow it to be used in constant and template declarations. Since the semantics of the `ref` depend on the store, this would mean that the semantics of all value constructors potentially depend on the store. Furthermore, it is not entirely clear what the semantics of the constant declared by

```
const strange = ref 1
```

should even be — should every use of `strange` in an expression result in a new element of storage being allocated, or should they all refer to the same address? We do not view this as a deficiency of our proposal, rather we view it as evidence that SML's treatment of `ref` as a constructor is a mistake.

One of the asymmetries of the design of SML is that one can define an injection-only view of a constructor (using a `val` specification), but not a projection-only view. As an example of the utility of such a mechanism, consider the implementation of *points* in a graphics toolkit. We may want to restrict points to some sub-range of the representable values, while still using pattern matching to decompose values. This might be done as follows

```
abstype pt_t = PTREP of (int * int)
with
  exception PtRange
  fun mkPt (a, b) =
    if (a and b are in range)
    then PTREP(a, b)
    else raise PtRange
  proj PT (x,y) = PTREP (x,y)
end
```

where the `proj` declaration defines an identifier that can be used only in patterns. It is safe to allow wildcards on the right-hand side of projection declarations. Because our framework separates the injection and projection aspects of constructors, it is fairly straightforward to add this kind of mechanism.

8 Implementation

We have built a simple testbed implementation of our proposal. At compile time, template identifiers are bound to (*trivpat*, *patexp*) pairs (the environment component of an injection or projection closure is coded in the representation choices for the constructors). When a known template occurs in a pattern, we inline expand it. Say that `C` is bound to (tp, pe) and we have the pattern p. We symbolically apply tp to p. This yields a substitution on the variables of tp (which are the same as the variables occurring in pe). Applying the substitution to pe yields a new pattern, which replaces `C` p.

For a functor parameterized by a structure with constructors, there are two implementation issues: what is the representation of the abstract constructors and how are they used in patterns. To handle the first question, we use implicit structure members for the injection and projection functions. Note that these only need to be added when the structure is made abstract (i.e., by functor application), and can be generated as part of signature thinning. The injection function is an ordinary function, while the projection is a function that returns either the sub-values or FAIL. When building a decision tree, the compiler treats abstract templates as ordinary constructors, which allows merging of matches against them. A test against an abstract constructor is a call to the projection function. Of course, the use of functions to implement the construction and destruction associated with abstract constructors may result in a certain degradation of performance. Of particular concern is the loss of merging when two abstract constructors have common structure. (For example,

the templates `LAMBDA` and `AP` defined earlier share the structure `TERM(OP _, _::_)`. Any value not matching this pattern cannot possibly match either of them). Other costs include the loss of inline tests and the replacement of branch tables with trees of conditionals. Note that these costs are incurred only when constructors are actually abstract, that is, when functors are used. In particular no penalty is incurred when structures, which have transparent signatures, are used rather than functors. Moreover if macro expansion were used in the implementation of functor application (a reasonable thing to do when compiling a production version of a system) functors would reduce to structures and abstract constructors would become concrete.

Our implementation of abstract constructors can also be applied to solve an outstanding problem with datatype representation and abstraction. In some implementations of SML [App90], the representation of the datatype defined by `datatype d = A | B of t`, depends on the representation of `t`. If the representation of `t` is appropriate, it is possible to represent the value `B` *exp* with the representation of *exp*. Problems arise when `t` is abstract, since its representation is known at the definition of `d` but not elsewhere. Extending our technique to cover abstract datatype constructors that fall into the danger zone solves this problem. Although our solution to the problem incurs a performance penalty, a less speedy program is better than one that does not run correctly.

9 Related work

Wadler's view mechanism ([Wad87]) shares the objective of allowing data abstraction and pattern matching to cohabit. Views were once part of the Haskell definition ([HW88]), but were dropped because of technical difficulties. Conceptually, a view of a type T, is a datatype T' together with a pair of mappings between T and T'. Ostensibly these maps are isomorphisms, but since they are defined by the user, there is no assurance that the types are truly isomorphic. Views and templates differ in several significant ways. The principal difference is that Wadler's views define maps between concrete representations, whereas templates provide abstract views of a single representation. Because views define different types, a given pattern match can involve only one view. In addition, once a view is defined, it is not possible to add additional constructors (even if other representations admit additional objects). Templates, on the other hand, do not suffer these restrictions. The implementation of views uses the user defined maps to convert between representations; thus, pattern matching can incur arbitrarily large performance penalties[2]. In our scheme, most uses of templates incur no run-time cost, and even in the worst case they add only a constant overhead to pattern matching. Our presentation of the semantics of templates is more detailed than that of views given in [Wad87]; furthermore, we address the semantic and implementation issues related to separate compilation and parameterized modules.

The CAML system ([WAL+]) provides a mechanism for defining new concrete syntax, by specifying a grammar to map quoted phrases to the internal representation of programs. This mechanism could be used to implement our template mechanism, although the imple-

[2] In fact, there is no guarantee that the maps even terminate.

mentation details appear non-trivial. Recently, a *quotation mechanism* has been proposed for SML, which allows terms in some object language to be included in expressions ([Sli91]). This provides some of the syntactic convenience of our mechanism, but it provides no help for pattern matching against terms of the object language.

References

[App90] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 4(3):343–380, November 1990.

[AR] William Aitken and John H. Reppy. Abstract data constructors — symbolic constants for Standard ML. Cornell University Technical Report (*in preperation*).

[HW88] Paul Hudak and Philip Wadler. Report on the functional programming language haskell (draft proposed standard). Technical Report YALEU/DCS/RR-666, Yale University, Department of Computer Science, December 1988.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, N.J., 2nd edition, 1988.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML.* The MIT Press, Cambridge, Mass, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Mass, 1990.

[Sli91] Conrad Slind. Varieties of object language embedding in standard ML, 1991. *unpublished.*

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 307–313, January 1987.

[WAL+] Pierre Weis, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. *The CAML Reference Manual (Version 2.6).* Projet Formel, INRIA-ENS.

Efficient representation of extensible records

Didier Rémy
INRIA-Rocquencourt*

April 10, 1992

Abstract

We describe a way of representing polymorphic extensible records in statically typed programming languages that optimizes memory allocation, access and creation, rather than polymorphic extension.

Introduction

Type systems for records have been studied extensively in recent years. New operations on records have been proposed such as polymorphic extension that builds a record from an older one without knowing its fields. Such operations are very powerful, and were not always provided as primitive constructs in untyped languages.

In comparison to the numerous results on the type theory of records, there has been less interest in their compilation. Many languages still have monomorphic operations on records, e.g. most implementations of ML [HMT90, Wei89, Ler90]. Others, that have more powerful records, use association list techniques, eventually improved by caching.

Safe untyped languages require that the presence of fields is checked before access. The use of association lists interleaves dynamic checking with access to fields. In strongly typed languages, the presence of fields is statically checked. Thus the representation of records by association lists performs superfluous runtime checks, and it seems that cheaper solutions could be found.

We propose a representation of records based on a simple perfect-hash coding of fields that allows access in constant time with only a few machine instructions which can be dropped to a single instruction whenever the set of fields of the record is statically known. Creation can be performed in time proportional to the size of the record, and allocates a vector of size the number of fields plus one. Only the polymorphic creation of records has to pay more in time and memory.

In section 1, the specificity of records as data product structures serves as an introduction to the conditions for which our representation will work in practice. The method is described in detail in section 2 as the encoding of partial functions from labels to values with finite domains. In section 3 we extend the method to records with defaults. These are total functions from labels to values that are constant almost everywhere. As an application we get safe standard records in an untyped language. In section 4, we discuss how to handle pathological cases in order to prevent bad behavior.

1 Records and their specificity

Records are product data structures. Each piece of information is stored with a key, more commonly called a label, that is used to retrieve the information. There is at most one value associated to a label. By *field* a pair noted $a \mapsto v$ of a label a and a value v. Such data structures are of common use in computer science. However our definition of records is too vague for choosing a good representation of records. It is necessary to know the average and the maximum size of these structures, to know whether they are created incrementally, the frequency of the different operations and which ones are privileged. It is obvious that different programs will give different answers. These questions can only be answered in general, or the answers that we give below can also be taken as assumptions for which our representation of records will work in practice.

Records are provided with three operations. *Creation* builds a record with a finite number of labels together with values associated to these labels. *Extension* takes a record and builds a new one that has all fields of the first one plus a finite number of new fields. *Access* takes a record and a label and returns the value associated to that label. It fails if the label is not in the domain of the record.

Records have a relatively small number of labels. At most a couple of hundred, on average less than ten. Non incremental creation and access are both very frequent and are privileged. There are usually many records with the same domain. Space and time are equally

*Author's address: INRIA, B.P. 105, F-78153 Le Chesnay Cedex. Email: Didier.Remy@inria.fr

important.

The simplest representation of records by association lists is good for very small records but it makes access to large records too slow. Balance trees would have better performances for large records but the overhead has also to be paid for small records; they also require too much memory. General hashed tables will also have an overhead that is not acceptable for small records.

2 Extensible records with polymorphic access

In this section we consider records as partial functions from labels to values, with finite domains. The problem is to find a representation for such functions, such that under the assumptions of the above section, the operations on records can be performed efficiently. By performing, we mean evaluating in general. In particular, this applies to compilation where some of the evaluation can be done statically.

Standard monomorphic operations on non extensible records should not be penalized by the introduction of more powerful records. Although it is always possible to keep two kinds of records coexisting, the new records should replace the older, weaker ones. It should be left to the compiler to recognize that some record operations are monomorphic and thus can better be compiled. Indeed, this will not be possible for all compilation schema.

A record r is a partial function from labels to values with a finite domain $(a_i \mapsto v_i)_{i\in[1,n]}$. The simplest decomposition of this function is

$$\text{Labels} \xrightarrow{h} [1,n] \xrightarrow{v} \text{Values}$$

$$a_i \longmapsto i \longmapsto v_i$$

The total function v stores the components of the record, while the the partial function h, called the header, describes how labels are mapped to indices. This decomposition suggests the representation of r in a vector R:

$$\begin{cases} 0 \mapsto H \\ i \mapsto v_i \quad i \in [1,n] \end{cases}$$

where H represents the function h.

The partial function h needs to be defined at least on the domain of r and it should better be injective on the domain of r too. Any such function would work, since v is then defined by

$$(h\restriction dom(r))^{-1} \circ r$$

up to permutation of indexes with identical values.

If all records are coded such that their headers (the representations may differ provided they implement the same function) only depend on their domains, then compilation of operations on records whose set of fields is statically known can be optimized by partially evaluating their header. The access become a single indirect read to fetch the value of the record on that field. The creation is always in that case, since it builds a record with n fields from nothing. The header can be computed statically and shared between all records built by the same function. The cost is reduced to allocating and filling $n+1$ fields of a vector. More generally, headers can be shared between all records that have the same domain by keeping all existing headers in a table.

Polymorphic access and polymorphic extension must use the headers. For sake of simplicity, we consider that labels are integers. The parser and the printer would deal with the isomorphism between integers and names in a real language.

Finding a good representation of h seems as difficult as finding a good representation of r. There are two differences, though. Since the header is shared, we are allowed a little more flexibility on the size of H. Also, h is a function on integers, thus we are allowed to use arithmetic and logic operation on integers. There is no hope of finding a direct representation of h by arithmetic operations, since its domain is completely arbitrary. At least some mapping between integers has to be an arbitrary map represented by a vector of integers for instance. A mixed decomposition of the header h is:

$$\mathbb{N} \xrightarrow{(\_ \bmod p)} [0,p-1] \xrightarrow{\eta} [1,n]$$

where $(\_ \bmod p)$ is injective on the domain of r. Such a decomposition is always possible, to the price of a higher p. In practice the smallest p that works is on average twice the size of n for a few labels and three to four times for larger sets of labels. Since the header is shared, this is very acceptable.

The interest of this decomposition of h is that it can be compiled efficiently and coded in a vector H:

$$\begin{cases} 0 \mapsto p \\ j \mapsto \bar{\eta}(j-1) \quad j \in [1,p] \end{cases}$$

The partial function η must be extended into a total function on $[1,p]$. We write it $\bar{\eta}$ the unconstrained extension of η.

In too cases below, We will also be interested in two particular extensions of η below that we write $\hat{\eta}$ and $\check{\eta}$. The former $\hat{\eta}$ is an extension of η with values of $[1,n]$, thus it makes r a total function. The later $\check{\eta}$ extends η outside of $[1,n]$, for instance 0, which provides a membership test to the domain of η by testing $\check{\eta}$ for equality to zero.

The domain of r must also be coded in H for polymorphic extension. All its labels can be listed at the end of H.

$$\begin{cases} 0 \mapsto p \\ 1 \mapsto n \\ 2+j \mapsto \bar{\eta}(j) & j \in [0,p-1] \\ 1+p+i \mapsto a_i & i \in [1,n] \end{cases}$$

The access can be optimized whenever the domain of the record, and consequently the header, are statically known. Such information is expected to be found by the typechecker. This cannot always be the case, however.

In order to rely on the types to know the domains of records, the attendances of fields, given by the types of records, must correspond exactly to their domains. This implies that the restriction of a record on a field modifies its header, since its changes the attendance. This is one possible semantics for restriction. Another one is to take the restriction of fields as a retyping function, that is, a function that evaluates as the identity. The choice is between an expensive active restriction that allows access optimizations or a cheap retyping restrictions that forbids them.

3 Records with defaults

In this section we consider records as partial functions from labels to values, constant almost everywhere. The problem is now to recognize whenever the field does not belong to the domain of the record (we mean the explicitly defined values, here), in which case the default value is returned. The membership test might be expensive in time or in space.

There is a cheap solution based on the same technique as above. In fact, we coded records by total function on labels, and described the domain separately in order to implement the extension of fields. Thus we could apply them outside of their domain (but get a value of unpredictable type).

Let r be a record. Consider the record r' equal to $id \restriction dom(r)$. An arbitrary label a is in the domain of r if and only if it is equal to $r'.a$. The auxiliary record r' only depends on the domain of r is already be coded in the header H as the domain of r. Remember that i is the index in R where the value of label a_i is stored. Thus it is the index where a_i is in R', which is also in H at position $2 + p + i$, provided $i = \eta(a_i \bmod p)$.

We simply shift the indices in R to place the default value at position 1:

$$\begin{cases} 0 \mapsto H \\ 1 \mapsto d \\ 1 + i \mapsto v_i \quad i \in [1, n] \end{cases}$$

We compute the application of r to the label a as follows. First compute the index i associated to a, that is $H.(a \bmod (H.0))$. If $H.(2 + p + i)$ is equal to a, then the label belongs to the domain of r and the result is $R.(1 + i)$ otherwise it is the default $R.(1)$.

One must be careful to use the $\hat{\eta}$ extension of η. The $\hat{\eta}$ extension is still possible, but the above membership test must be preceded by a membership test to the domain of η, and in case of failure the default value should also be used.

In fact, the encoding of records with defaults can be easily adapted to implement safe classical records in an untyped language: a dynamic type error is raised when the membership test fails instead of returning the default.

n	3	5	8	11	23	30	40	60	100	200
p_{ave}	4	9	18	30	86	132	207	400	902	2565
p_{max}	11	18	29	48	148	206	298	576	1195	3053

Figure 1: Average header size

4 About efficiency

This section does not consider the case of records with defaults for sake of simplicity, but it can be adapted very easily to them.

The previous representation of records is very attractive since it implements linear time access, uses very little memory, keeps the same performance on monomorphic records as if all records were monomorphic. However we must check the following points. First, that the size of the header does not get too large. Secondly, that the polymorphic extension does not have too bad performance, even though it is not privileged. Last, that pathological cases can be handled.

The computation of the integer p is at the heart of every question. The problem is given a set of integers D, find a small integer p such that $(\_ \bmod n)$ is injective on D. The integer p does not need to be minimal, even if computed at compile time, since a larger header might make polymorphic extension more efficient. However, the minimal p gives a lower bound on the size of the header. For instance if D is randomly chosen, and the set of labels is large enough in comparison to the size n of D, the probability that p disambiguates n integers is

$$\frac{p!}{(p-b)!\, p^n}$$

The formula that gives the average smallest p in function of n is simple, but figure 1 gives an experimental result on the average p_{ave} and the largest p_{max} for on a hundred runs per column.

For small records, 10^4 runs did not give very different maximal p. The dispersion of p is shown by figure 2

The figures show that under 30 labels, the size of D does not exceeds, in average, four times the number of fields, and exceeds very rarely twice the average. For large records (above 50 fields) the header becomes very large. It is clear that another solution must be applied for large records. Even if one wished to push this limit to a hundred labels, there is always a rank that can be reached in practice (even if it is pathological) for another representation should be used.

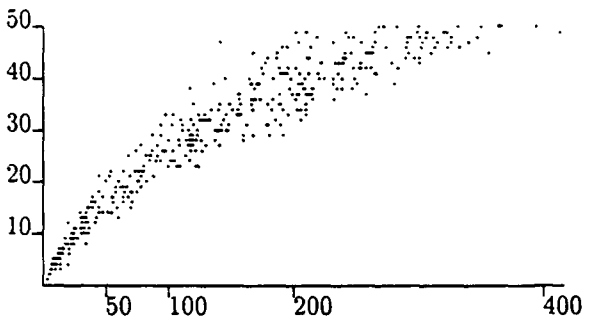


Figure 2: Dispersion of the header size

Since, we cannot avoid a mixed representation, if we want to handle large size records in order to represent them with reasonable size headers, we use tags to distinguish between the two representations. For instance, negative integers can be used as tags for an alternate representations. There are many possibilities for the alternate representations, and since these are pathological cases, in the sense that they do not meet the requirement that we set in section 1, we do not care much about the efficiency of the alternate representation. We propose, two possible solutions that fit well with the regular representation.

The first solution is whenever a and b are equal to j modulo p to assign $\eta(j)$ with an integer $-q$ such that q is in $[2,p]$ and modulo q distinguishes a and b and with values that are not in the image of η.

The second solution replaces perfect hashing by hashing with linear probing ([Sed88], Chapter 16). The header is H is

$$\begin{cases} 0 \mapsto p & \\ 1 \mapsto n & \\ 2+j \mapsto \bar{\eta}(j) & j \in [0,p-1] \\ 1+p+i \mapsto a_i & i \in [1,n] \\ 2+n+p+j \mapsto a_{\eta(j)} & j \in [0,p-1] \end{cases}$$

for labels that do not conflict. When $2+n+p+(a_i \bmod p)$ is already occupied, the label a_i is placed at the smallest free position after, say j, and $\eta(j)$ is filled with i.

The first method still gives access in linear time, but headers are more difficult to compute. The may be much faster on average, but require larger headers. In both cases there is a lot of freedom on how to chose p, according to how many conflicts are accepted. Letting p be about 3 times the size of r may avoid searching for optima while limiting the number of clashes. The first method is more flexible, since a conflict for one label does not need to double the size of the header or recompute another header: it simply uses the holes of the actual header. Then, it can also be used even for average size records in some cases in order to compile more efficient extension.

The efficiency of polymorphic extension has not been considered yet, since it was not a privileged operation. It has to dynamically compute a new header, which may be very expensive — and at least proportional to the size of the record it extends. The time for computing the header now becomes important. There are different cases (we exclude large size records, that should be represented otherwise):

- There is no need to compute a new p,
- The average case for computing a new p,
- The worse cases for computing a new p.

In the average case, computing a new header means that 3 different p's must be probed per extension, since headers are in average 3 times the size of n. The optimistic unit cost U is the one of a loop that contains at least one vector write and one modulo instruction. The cost for a probe is pU, since the failure is probable to happen at the end. Thus the average cost for creating a new header is $3pU$ plus two other pU for filling the header.

However, a record with a very compact header may be extended with a label that will make the header get closer to its average size. Then about nU probes may be needed, making the cost for the new header increase to pnU.

On the other hand, it is very probable that the extended header already exists or is trivial. If the label a of the extended field is taken at random, there is a probability of $(p-n)/p$ that the $n+1$ fields will still be disambiguated by p. In that case, the cost for creating a new header is the same as copying the old one plus modifying a few fields pU.

The creation of a new header may be avoided most of the time since it is very probable that it has already be created. This requires that all existing headers are stored in a dictionary. This will save both space and time. The keys in the dictionary are the domains of headers that can be kept ordered in H, so that equality tests are not too expensive, and the whole cost of searching would be lower than the minimal cost of extension.

Active restriction of fields can be implementing along the same ideas. The header is looked up in the table. If it does not exist, then the new header need not be the smallest one, provided that it is of reasonable size. This avoids the expensive cost of finding the smallest integer modulo which all elements of the domain as distinguished.

5 Other compilation schemas

There are three different ideas in the above representation of records

1. The value of fields and the position of fields are represented separately. The header that describes the later is shared between all records having the

same set of fields, two records with the same domain always have the same fields at the same position.

2. The header can be represented by a modulo followed by a projection.

3. Different representation of the headers can live together.

The first point is crucial in our representation. Any representation of records that does not originally respect this point can still be used to implement headers, then values can be stored in vectors as above. Sharing of headers will save the large amount of space required by association lists or balanced trees.

The representation of the header itself is not important. We described one possibility that is very convenient for small and medium size records. But many other representations are possible. We choose to represent the header as a structure that is interpreted both by extension and access. It could also be a closure for the access part, together with a description of the domain that is needed for the extension. This is the tag vs closure duality.

If the extension and the restriction of fields are themselves closed with the header, records could really be viewed as objects with two methods for access and extension.

Conclusion

We have presented a way of representing records with or without defaults that allows efficient access and creation. Only the more powerful features such as polymorphic extension, or true restriction of fields have to pay a higher price.

Our representation of record with defaults can be used to implement safe access in an untyped language.

An orthogonal application could be the representation of feature terms that are very related to records.

Thanks

These ideas originate in discussions with Xavier Leroy. They have been mentioned for the first time in [Ler90] and tested in the untyped version of Zinc, the ancestor of Caml-Light.

References

[HMT90] Robert Harper, Robin Milner, and Mads Tofte. *The definition of Standard ML*. The MIT Press, 1990.

[Ler90] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA-Rocquencourt, 1990.

[Sed88] Robert Sedgewick. *Algorithms*. Computer Science. Addison-Wesley, second edition edition, 1988.

[Wei89] Pierre Weis. *The CAML Reference Manual*. BP 105, F-78 153 Le Chesnay Cedex, France, 1989.

A General and Practical Approach to Concrete Syntax Objects within ML†

Mikael Pettersson and Peter Fritzson

Department of Computer and Information Science, Linköping University
S-58183 Linköping, Sweden
Email: mpe@ida.liu.se, paf@ida.liu.se

Abstract: In this paper we present an approach to concrete syntax object within ML, which is both general and efficiently implementable. These language enhancements add BNF rules for abstract syntax declarations and "*semantic brackets*" [| ... |] with inline concrete syntax and pattern matching for readability and conciseness. This approach has several improvements integrated together which either do not appear in previous works, or appear in forms which are very restrictive or have very inefficient implementations. Our improvements are: (1) inline concrete syntax within "*semantic brackets*" has been integrated both with the ML type system and the ML scope rules, (2) concrete syntax can appear both as *syntactic patterns* for pattern matching and as *syntactic expressions* for building objects, (3) patterns can be nested to arbitrary depth, (4) concrete syntax and ML objects can be mixed; so called *anti-quotations* are supported directly, (5) patterns and parts of patterns can be augmented with type information, (6) efficient integration with a general incremental LR(1) parsing mechanism.

These extensions have been efficiently implemented within our DML system. DML, the Denotational Meta-Language, is a dialect of Standard ML with extension aimed at making it (even more) useful for implementing denotational specifications of programming languages.

1. Introduction

Functional languages generally have some facilities for defining and computing with inductively defined data types. Unfortunately, only the simplest prefix syntax is available for objects of these types. It is generally true for problem domains where data objects with syntactic structure are specified and later accessed and manipulated, i.e. when a *meta-language* computes with an *object-language*, that the implementation tends to be cluttered with prefix constructors and/or access functions for abstract syntax and other intermediate forms. Typical examples are conventional compilers and implementations of denotational specifications of programming language semantics. If context-free grammatical methods can be used to specify this structure, then a set of language extensions based on inline concrete syntax, e.g. as proposed in this paper, can lead to more compact and readable code. The language extensions described in this paper, implemented within our DML system, have so far been used primarily in compiler generation from denotational semantics specifications. An overview of the system is given in [PF92], while this paper concentrates on the

† This work was supported by the Swedish National Board for Industrial and Technical Development (NUTEK).

implementation details of the concrete syntax extensions.

2. Related Work

The work by Aasa *et al* [APS88] describes an approach to inline syntax, which in its current form only applies to sequences of characters and is so inefficient that it is not practically usable. Its inefficiency is mainly due to its use of Earley's [Earley70] algorithm which repeatedly recomputes parsing automatons which are reused in our method. An implementation has been done, hidden within the LML (Lazy ML) system, but it is not used by LML itself [Augustsson91].

Lee [Lee89] has a very simple but restrictive approach to implementing in-line syntax within semantic brackets in the context of denotational semantics specifications. A syntactic pattern is converted to a string by simply concatenating everything between the semantic brackets. A similar conversion occurs for each production in the syntax declaration. Patterns in semantic equations are then resolved by searching linearly for matching strings from the set of productions. This disallows patterns with different variable names but similar structure, and patterns with nested structure. These restrictions are not present in the DML system.

The Refine language [Refine90] contains perhaps the most practical and complete implementation of inline concrete syntax before our work, but still lacks integration with a general type inference mechanism and often has problems with parsing conflicts due to its use of standard LALR(1) techniques.

The CAML system from INRIA [Weis *et al* 89] also includes a facility that can be used to provide concrete syntax for datatypes. In CAML, a `grammar` declaration consists of a set of productions, each associated with an ML expression that builds the appropriate resulting term. The grammar is sent to Yacc, and the parsing tables are read back in and mapped to a set of parsing functions. These functions can be applied to strings, files or the current input. When parsing, the input is pre-tokenized by the ML system into a stream of identifiers, strings and numerical constants. It is possible to mix concrete syntax and ML expressions (so called *anti-quotations*) since the system's parser for ML expression is visible as the function `parse_caml_expr0`, and can be called from the user's code in the grammar rules. The special syntax <<...>> is used to delimit parts of an ML program that should be parsed according to user-defined grammar. There is a notion of *the current grammar* which is used by default; <:*grammar*:*non-terminal*<...>> can be used to explicitly direct the focus to a particular grammar and (optionally) a particular start symbol.

The system appears to be most suitable for providing a simple means to prototype parsers for the concrete syntax of programs when developing compilers or interpreters. Since it runs entirely before the type-inference process, no ambiguities can be allowed, even if type constraints would resolve them. Overall, the system has a more "programmatic" than declarative feel. While our approach doesn't support the building of user-level parsing functions, we think it is better integrated with the ML language, especially since it uses type information *in addition to* context-free syntax, and from the start supports anti-quotations without the need for extra programming.

3. The Use of Syntax Definitions in Denotational Semantics

A common way to specify the type of syntax trees in texts on denotational semantics is to use BNF rules like this:

```
C : Con
I : Ide
E : Exp
E ::= C | I | E + E
```

and then refer to syntax tree objects by writing their syntax between "semantic brackets":

```
eval ⟦ C ⟧ env = C
eval ⟦ I ⟧ env = env I
eval ⟦ E1 + E2 ⟧ env = (eval E1 env) + (eval E2 env)
```

The definition-by-syntax facility is really just a shorthand, eliminating the need to explicitly deal with Cartesian products, disjoint unions and their appropriate injections, projections and tag tests. In DML we take the same view: syntactically defined types provide a convenient notation for ordinary types defined with `datatype` declarations. The example could in DML be expressed as follows:

```
type Con = ...
type Ide = ...
syntax Exp = Con | Ide | Exp "+" Exp

fun eval [| c |] env = c
  | eval [| i |] env = env i
  | eval [| e1 "+" e2 |] = (eval e1 env) + (eval e2 env)
```

Translating it to vanilla SML might result in:

```
datatype exp = ConExp of Con
             | IdeExp of Ide
             | AddExp of Exp * Exp

fun eval (ConExp c) env       = c
  | eval (IdeExp i) env       = env i
  | eval (AddExp(e1, e2)) env = (eval e1 env) + (eval e2 env)
```

The transformation is non-trivial in general. In this example, the "pure" SML version wasn't too ugly, but more complex types and patterns (especially nested ones) are clearly more conveniently handled using grammar rules. There are two basic reasons for preferring syntactic type definitions. First, we do not have to cast all expressions and patterns into the prefix or binary infix syntax of SML. Instead we can use (almost) any meaningful notation we wish. Second, grammars allow us to make "unit productions" (type conversions) implicit. Consider:

```
(* DML version *)
fun simplify [| e "+" 0 |] = e
  | ...

(* SML version *)
fun simplify (AddExp(e, ConExp(IntCon 0))) = e
  | ...
```

Here DML enables us to make the mapping from integer constants to `Exp`s implicit, whereas SML wants us to spell out all the gory details.

4. The implementation of DML's syntactic objects

This section describes in detail how the extensions for syntactic objects are implemented. First the facility itself is presented, and an example is given to show that syntax alone cannot be used to resolve syntactic objects; types are needed as well. Then the compiling strategy is outlined, followed by detailed descriptions on how the pseudo-parallel LR(0) automatons used here are defined and implemented.

4.1 Grammar of the DML `syntax` extensions

Table 1 contains the grammar for the DML syntax extensions in the same style as the Standard ML definition [HMT90]. We use *string* to denote the class of all string constants, whereas the definition of Standard ML conflates all constants to the class *scon*. <*thing*> means that *thing* is optional.

Table 1: Grammar of the DML extensions for syntactic objects

dec	::=	**syntax** *synbind* <**withtype** *typbind*>
synbind	::=	*tyvarseq tycon* = *srules* <**and** *synbind*>
srules	::=	<*srule*> <| *srules*>
srule	::=	*string* <*srule*>
	|	*ty* <*srule*>
atexp	::=	[| <*synexp*> |]
synexp	::=	*string* <*synexp*>
	|	*atexp* <*synexp*>
atpat	::=	[| <*synpat*> |]
synpat	::=	*string* <*synpat*>
	|	*atpat* <*synpat*>

4.2 Syntactic ambiguities

Purely syntactic preprocessing is often not enough to determine the constructor form of a syntactic expression or pattern. Syntactically ambiguous patterns are common in denotational semantics specifications.

There, implicitly-typed variables are used to disambiguate them.

Syntactic Domains:

N	$\in$	**Nml**	(numerals)
I	$\in$	**Ide**	(identifiers)
E	$\in$	**Exp**	(expressions)

Abstract Syntax:

$E \quad ::= \quad N \mid I \mid E_1 + E_2$

$\mathrm{E} : \mathrm{Exp} \rightarrow \mathrm{Env} \rightarrow \mathrm{Int}$
$\mathrm{N} : \mathrm{Nml} \rightarrow \mathrm{Int}$

$$\begin{aligned} \mathrm{E}[\![N]\!]\rho &= \mathrm{N}[\![N]\!] \\ \mathrm{E}[\![I]\!]\rho &= \rho[\![I]\!] \\ \mathrm{E}[\![E_1 + E_2]\!]\rho &= \mathrm{E}[\![E_1]\!] + \mathrm{E}[\![E_2]\!] \end{aligned}$$

Here the implicit typing of N and I act to disambiguate the first two patterns. In this work, we wanted for syntactic patterns to behave as much as possible like ordinary SML patterns *without* having to introduce additional declarations or limit the way patterns may be formed (as in Lee's MESS system).

DML would in the first equation find that the type for N had been constrained to be a numeral, and in the second equation that I had to be an identifier. For this example this is enough to completely resolve the apparent ambiguities. In those cases where the implicit type constraints aren't enough, the ordinary SML *pattern* construct can be used to explicitly add the needed constraints (much like how overloaded variables sometimes need this). Note also that this problem is largely eliminated with the use of "noise" keywords in `syntax` productions.

4.3 General Implementation Strategy

1. `syntax` is seen as a grammatical alternative to ordinary `datatype` declarations. In the same vain, we see syntactic expressions and patterns as alternatives to ordinary expressions and patterns built using constructor applications.
2. Every `syntax` declaration is implicitly transformed to a a `datatype` declaration, derived as follows:
   Given a production rule *srule*, its constructor name is built by appending the name of the non-terminal (the `tycon`), a string "::=", and the names of the items in the right-hand side. The name of a terminal is itself, the name of a type is "_". For the same rule *srule*, the arity of the constructor is determined by building a tuple type `ty` of the embedded types (not the terminals) in the right-hand side. If `ty` is the empty tuple type, then the constructor is a constant, otherwise it is a function with domain `ty`. For example:

```
syntax Stmt ::= "if" Exp "then" Stmt "else" Stmt
              | "skip"
```

has the following corresponding `datatype` declaration:

```
datatype Stmt = "Stmt::=if_then_else_" of Exp*Stmt*Stmt
              | "Stmt::=skip"
```

Just as for `datatype` declarations, it is an error for a `synbind` to bind the same `tycon` or (generated) constructor more than once.

3. Occurrences of syntactic objects are translated to ordinary objects (expressions or patterns) built using explicit constructor applications. For every `syntax` declaration, its productions are added to an incrementally-built pseudo-parallel LR(0) parser[1]. Each syntactic object is first parsed by the pseudo-parallel LR(0) automaton which results in a set of *possible* parse trees. When all surrounding type constraints are known (at the end of the type-checking pass) only those trees that are consistent with later type assumptions are considered. If only one such tree remains, then that is taken as the meaning of the syntactic object.
4. Given that `syntax` declarations has had their corresponding `datatype` declarations determined, and that all *synexps* and *synpats* has been rewritten to ordinary expressions and patterns, we see that no extra constructs remain for the code generator. The run-time costs of these syntactic constructs is therefore the same as if ordinary datatypes had been used.

4.4 Resolving Syntactic Objects

This section explains how syntactic objects are parsed and rewritten to ordinary SML constructs. This process is integrated with the ordinary type checking process since the two are interdependent.

The static context C is extended with a grammatical environment `GrmEnv`. The idea is that a grammar is a finite mapping from constructed types (`ConsTypes`) to finite sets of productions, where each production is a sequence of terminals (string constants) and non-terminals (types). Grammatical environments follow the same scope rules as the type environments from `datatype` declarations.

$$\begin{array}{lcl} \text{GrmEnv} & = & \text{ConsType} \rightarrow \text{setof(Prod)} \\ \text{Prod} & = & \text{Token*} \\ \text{Token} & = & \text{String} \cup \text{Type} \end{array}$$

4.4.1 LR(0) construction

Given a context C, an LR(0) parser for the grammar environment in C is constructed as follows. First the LR(0) sets of items are generated for the grammar in C, where every bound `ConsType` is a valid start "symbol". This is achieved by introducing a dummy start symbol S', and adding the production S'→ty, for every bound `ConsType` ty. Non-grammatical types are effectively treated as terminals.

---

[1] These machines have the recognizing power of LR(1).

Then the transitions of the parser are defined by the following rules, which are based on the standard ones in [HU79][2]. Note that the rules do *not* disallow shift/reduce or reduce/reduce conflicts. We define a parser configuration as a 4-tuple (*pstk, astk, σ, input*). This is just the normal LR-parsing configuration extended with a result stack and a type substitution.

- *pstk* is a stack of state numbers (the usual parse stack)
- *astk* is a stack of abstract syntax trees (the "result" stack)
- σ is a finite substitution from type variables to types
- *input* is the remaining sequence of input tokens, terminated by the special terminal $. Each token is either a terminal string, or a pair (ty,x) where x is an ML syntax object (an Exp or a Pat), and ty is its type.

INIT: $([q_0], [], \emptyset, input)$
Init is the initial configuration, where q_0 is the start state of the automaton.

ACCEPT: $([q, q_0], [ast], \sigma, [\$])$
Accept is an accepting configuration if q contains a complete item S'→ty•. The result is *ast*.

SHIFT: $(q::qs, astk, \sigma, str::input) \Rightarrow (q'::q::qs, astk, \sigma, input)$
Shift if there is a transition labeled by the terminal *str* from q to q'.

GOTO: $(q::qs, astk, \sigma, (ty,x)::input) \Rightarrow (q'::q::qs, LEAF(\sigma'(ty),x)::astk, \sigma', input)$
Goto if there is a transition labeled by some type ty' from q to q', $Unify(\sigma(ty),\sigma(ty'))$ succeeds with most general unifier σ', and σ' is the composition of σ and σ'.

REDUCE: $(q_1::\cdots::q_k::qs, ast_1::\cdots::ast_n::astk, \sigma, input)$
$\Rightarrow (q'::qs, NODE(\sigma(ty),con,[a_1,\cdots,a_n])::astk, \sigma, input)$
Reduce if q_1 contains a complete item ty→$prod$•, $length(prod)=k$, the number of types in *prod* (its arity) is n ($n \le k$), and *con* is the name of the constructor for this production.

The resulting parse trees are described by the following type:

```
datatype 'a Ast = LEAF of Type * 'a
                | NODE of Type * Con * 'a Ast list
```

Leaf nodes correspond to inserted ML objects (*anti-quotations*), while internal nodes correspond to the application of some (syntax) constructor to a sequence of sub-trees. Each node is also tagged with its type. The type parameter `'a` is the compiler's type of the leaf ML objects, corresponding to expressions or patterns.

4.4.2 Modifications to the Basic Type-Checker

The DML type-checker consists of two major components: the ordinary type-checker for the SML subset, and the new combined parser/type-checker for syntactic objects. The type-checker for the SML subset is implemented by combining the general structure and rules of the formal definition of Standard ML

[2] Actually, what we are constructing is a viable-prefix recognizing NFA for the LR(0) items.

[HMT90], and an imperative implementation of type unification and treatment of polymorphic types in [Cardelli87]. It is organized in two passes: Pass 1 is the ordinary top-down traversal which maintains type environments and checks type compatibility by unification. Occurrences of flexible record patterns and overloaded variables are remembered in a separate set of cleanup actions, since they in general aren't fully determined at that point in the traversal. Pass 2 finally iterates through the set and checks that all flexible record patterns and overloaded variables have been fully determined by type assumptions made later in Pass 1.

To deal with syntactic objects, the following extensions were made:

Pass 1 calls the syntax parser every time a syntactic object is encountered. If the resulting parse forest is ambiguous, then the occurrence is remembered in the set of cleanup actions and the syntactic object is assumed to have an unconstrained type α. Otherwise, the unambiguous result is patched into the ML syntax tree directly and its type is used as the type of the syntactic object.

Pass 2 tries to disambiguate remembered parse forests by a filtering process: any parse tree whose type is inconsistent with type assumptions made later in Pass 1 is removed. If only one tree remains, it is patched into the ML syntax tree. Otherwise, there is an error (ambiguity or inconsistency).

4.4.3 Incremental generation of the automatons

Since constructing the LR(0) automaton is a costly process, we do not want to waste time by redoing this each and every time a syntactic object needs to be parsed. The grammar environment in the static context is therefore accompanied by a cached, partially-built automaton that exactly mirrors the set of productions that currently are in the grammar. Scope changes and declarations that modify the grammar also update the automaton.

In [HKR89] a simple algorithm for *incrementally* updating the LR(0) automaton is given. Suppose that we are using the graph representation of the viable-prefix recognizing DFA. The idea is that each state is represented by its set of *kernel* items (see [ASU86]), the set of reductions, the sets of possible goto and shift transitions, and a "dirty" flag. Initially, only the start state exists, and it is "dirty". When the parser executes and wants to examine the transitions from a particular state, it first checks the dirty-flag and if set, computes the closure of the state. This in turn may cause new initial "dirty" states to be created, or links to be made to existing states.

The incrementality here means that changes to the grammar only need to do minor adjustments to the cached automaton. It is not until syntactic objects are being used that the real generation work begins. Once completed, the automaton is reused until further changes to the grammar causes new incremental updates.

4.4.4 Pseudo-parallel LR(0) parsing

It is generally believed that LR(1) or some of its approximations is needed to build deterministic parsers for standard programming languages. Unfortunately, LR(1) automatons are considerably more expensive to build than LR(0) automatons. A technique known as *pseudo-parallel* LR parsing was first developed for natural language processing by Tomita [Tomita85]. It achieves the recognizing power of LR(1) by running several LR(0) parsers in parallel. The idea is this: suppose that an LR(0) automaton in a certain state finds

some shift/reduce or reduce/reduce conflicts. Instead of giving up, it can be cloned, and each copy can be set off to pursue one of the alternatives. In practise, a single LR-engine maintains a set of parsers synchronized by all being at the same position in the input.

This approach has two advantages. First, the overhead of pseudo-parallel parsing is low enough to be practically useful when short sentences are involved (within a factor of two compared with LALR(1) on sentences with up to several hundred tokens [Rekers92]). Secondly, the method automatically handles the case when ambiguous grammars are involved, and the system wants to find *all* the parse trees rather than a single unique one.

The DML system uses both the incremental and pseudo-parallel methods. The latter fits nicely with DML's way of first generating all parse trees, and then removing the type-inconsistent ones. Currently the algorithm is based on [HKR89] which limits grammars to the "finitely ambiguous" context-free grammars (there must be finitely many parse trees for any given sentence).

4.4.5 LR(0) versus SLR(1) and LALR(1)

Why LR(0)? Would not SLR(1) or LALR(1) be more efficient? Not necessarily. The advantage of LR(0) is that no lookahead information needs to be computed. The disadvantage is that lookahead could be used to prune the number of generated parsers, instead of blindly trying all possibilities (in the case of shift/reduce or reduce/reduce conflicts). However, in [Lankhorst91] it is reported that SLR(1) and LALR(1) only buy about 10% in parsing time over LR(0). Note also that lookahead would have to be recomputed in some cases, as in the example below:

```
syntax  t1 = "foo"
and     t2 = "(" t1 ")"          (* FOLLOW(t1) = {")"} *)
let
  syntax t3 = "x" t1 "x" | "z" (* FOLLOW(t1) = {")", "x"} *)
in
  ...
end                              (* now, FOLLOW(t1) = {")"} again *)
```

5. Interaction with the ML module system

`Syntax` declarations have an unfortunate interaction with the module system. Ordinary `datatype` constructors can be accessed either by opening a module, or by using the long identifier syntax <module>.<name>. Since there is no way to refer to the constructors of a `syntax` declaration except by using syntactic objects, one *must* open the module in order to bring the constructors and the grammar environment in scope. An alternative would be to allow the user to explicitly name the constructors, for example:

```
syntax t = "foo"        : Foo
         | "bar"        : Bar
```

This alternative solution has not been investigated yet.

6. Conclusions

To our knowledge this is the first both general and efficient approach to concrete syntax objects within ML. An implementation, as presented in this paper, has been done within the DML system. The primary applications so far has been in denotational language specifications, including a C subset and full Scheme, from which compilers and interpreters have been generated. Traditional compilers not based on formal semantics have also been implemented. In all of these cases, the use of syntactic objects (mainly patterns) for instead of conventional `datatype`:s was found to make the specifications easier to read and write. Some care must be taken with highly ambiguous grammars; in these cases some strategically placed explicit type constraints always suffice to make the resolution successful. The implementation, although being a prototype, has been fast enough not to present any problems.

7. References

[APS88] Annika Aasa, Kent Petersson, and Dan Synek. *Concrete Syntax for Data Objects in Functional Languages* (Proceedings of the 1988 ACM Conference on LISP and Functional Programming, pp. 96-105, July 1988).

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques and Tools* (Addison-Wesley, 1986).

[Augustsson91] Lennart Augustsson. *Private communications*. (November 1991).

[Cardelli87] Luca Cardelli. *Basic Polymorphic Typechecking* (Science of Computer Programming, 8 (1987), North-Holland).

[Earley70] Jay Earley. *An Efficient Context-Free Parsing Algorithm* (Communications of the ACM, Vol. 13, No. 2, 1970).

[HKR89] Jan Heering, Paul Klint, and Jan Rekers. *Incremental Generation of Parsers* (Proceedings SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN NOTICES Vol. 24, No. 7, July 1989).

[HMT90] Robert Harper, Robin Milner, and Mads Tofte. *The Definition of Standard ML* (The MIT Press, 1990).

[HU79] John E. Hopcroft, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, 1979).

[Lankhorst91] M.M. Lankhorst. *An empirical comparison of generalized LR tables*. (Proceedings of the workshop on Tomita's Algorithm — Extensions and Applications, University of Twente, The Netherlands, 1991).

[Lee89] Peter Lee. *The Automatic Generation of Realistic Compilers from High-level Semantic Descriptions* (the MIT Press, 1989).

[PF92] Mikael Pettersson and Peter Fritzson. *DML - A Meta-Language and System for the Generation of Practical and Efficient Compilers from Denotational Specifications* (Proc. of

4th IEEE International Conference on Computer Languages, ICCL'92, 1992).

[Refine90] *Refine 3.0 User's Manual.* (Reasoning Systems Inc., Palo Alto, CA, USA).

[Rekers92] Jan Rekers. *Parser Generation for Interactive Environments* (Ph.D. thesis, University of Amsterdam, The Netherlands, 1992).

[Tomita85] Masuru Tomita. *An Efficient Context-free Parsing Algorithm For Natural Languages* (Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1985).

[Weis *et al* 89] Pierre Weis *et al. The CAML Reference Manual* (INRIA Technical Report no. 121, Version 2.6, 1989).

An optimizing ML to C compiler

Régis Cridlig

Ecole Normale Supérieure *
& INRIA-Rocquencourt

Abstract

Since the C language is a machine independent low-level language, it is well-suited as a portable target language for the implementation of higher-order programming languages. This paper presents an efficient C code generator for Caml-Light, a variant of CAML designed at INRIA. Fundamentally, the compilation technique consists of translating ML code via an intermediate language named Sqil and the runtime system relies on a new conservative garbage collector. This scheme produces at the same time excellent performance and good portability.

1 Introduction

ML is a complicated programming language. Its syntax is complex and its dynamic semantics is related to the semantics of lexical Lisps like Scheme (cf [IEE90]). Furthermore it offers a strong polymorphic type system and a safe module system. Because of its overall qualities this language has become more and more popular for research and teaching, but with the emergence of better compilers it could reach a larger audience in industry as well.

At this time there are a few good implementations of ML, but all suffer at least from one or another drawback: disappointing execution speed, lack of portability or excessive use of memory.

The existence of a static type system in ML should permit us to compile this language nearly as efficiently as Pascal (not quite because polymorphism requires all objects to be the same size). The choice of C [KR88] as target language allows us to get a portable compiler easily because C is so widespread and now well standardized; nevertheless this choice must not cause bad general performance. Using the K2 compilation kit of Nitsan Séniak and Vincent Delacour, we built a very efficient and machine-independent ML to C compiler [Cri91].

In this article we explain our approach to ML compilation, provide a sketch of our implementation and give some results. We use the word 'global' to mean imported or exported and its contrary 'local' to mean local within a given module or function.

*Address: Laboratoire d'Informatique de l'École Normale Supérieure - 45 rue d'Ulm, 75230 Paris Cedex 05, France. Electronic mail: Regis.Cridlig@ens.fr

2 Compiling strategy

We use a new compilation technique, named *explicit specialization* [Sén89], to obtain very good optimization of function calls: the cost of function calls is indeed the bottleneck inherent to the compilation of all functional languages.

Our compiler reduces the semantic complexity of ML programs, according to some simple and well-defined steps, to a level acceptable by a compiler with a familiar technology like a typical C one. As a matter of fact people are accustomed to writing programs that use mostly the capabilities of classical algorithmic languages such as Pascal. In this case, the dynamic usage of local functions is limited to the static scope of their definitions, which fortunately allows us to handle functional values without allocating a closure in the heap: well-known and efficient techniques (cf [ASU86]) can be used to compile classical algorithmic languages.

Since data is handled in the obvious manner in our translation scheme and is later optimized sufficiently by modern C compilers, we concentrate our efforts and description on control flow.

2.1 The intermediate language: Sqil

The specialization techn consists of translating ML programs that use general control flow constructs into more ecialized and better compilable forms, the Sqil ones.

The Sqil dialect, kernel of the K2 compilation kit, is a kind of lexical and bivalued[1] Lisp with continuations and functional values restricted to the global environment. Since Sqil only addresses the efficient compilation of control flow, it is limited to a few elementary special forms:

- `defun`, `defvar`, `progn`, `let`, `if`, `setq`, `labels` and `flet` are present with the same meaning as in Common Lisp (cf [Ste84]);
- (`function` *symbol*) yields the functional value of *symbol*, which must be a global function. This value can be later used in a computed call with `funcall`;
- (`the-continuation`) yields the calling continuation of the global current function definition. This continuation can be later invoked by (`continue` *cont value*);
- finally, (`block` *name expressions...*) and (`return-from` *name value*) build lexical escapes.

The restriction on functional values and continuations guarantees one important property: identity between variables and registers (see [Sén91]), meaning that each local variable of a program can be allocated to a register. Additionally function call is direct by default thanks to bivaluation; computed calls explicitly use `funcall`.

We successfully used the K2 compiler written by Nitsan Séniak, who has developed original techniques to compile local functions towards C [Sén90]:

- *Displacement of continuations* propagates calling continuations of functions into the called functions in order to transform many non-tail calls into tail ones, which can be compiled using `goto`.

[1] I.e. symbols can have a functional value and a non-functional one at the same time, like in Common Lisp.

- *Function integration* in another function definition allows K2 to compile mutually tail calls between several functions into ordinary jumps.

The K2 compiler has proved to yield very good C code. Nevertheless Sqil could be directly compiled to efficient native code as well.

2.2 C as a target language

The Common Lisp compiler AKCL [Sch88] shows that it is realistic to compile Lisp or ML towards C used as a portable target language. Actually the primary advantages of C are:

- C is a low-level language, whose compilation captures all previously performed high-level optimizations well.
- C is highly portable and available on an increasing number of architectures, partly thanks to the spread of the Unix system.
- Thirdly, C compilers progress each year, while good assembly code generation by hand for RISC architectures is becoming very complicated.

However, in some respects, C is not so well-suited as a target language. Here are some reasons:

- The compiler designer loses full knowledge and mastery about the generated machine code: for instance, some critical optimizations concerning the function calling sequence that are performed by some Lisp compilers cannot be carried out any more. Moreover multiple results can be inefficiently compiled.
- Some specialized machine code instructions cannot be generated by the C compiler, like arithmetic operations with overflow checks. But this is less relevant for RISC architectures.
- Moreover, the arrangement of local variables in the C stack or in registers is left undefined. We cannot therefore use a classical garbage collector because it requires us to identify precisely all roots that can point to allocated data. Consequently we shall use a more complicated conservative garbage collector with ambiguous roots.

3 Implementation

3.1 The front-end

Our front-end is much the same as ZINC's one, written by Xavier Leroy [Ler90]. It first parses the Caml-Light grammar and then infers the types of the expressions using a variant of Milner's algorithm [Mil78]. The typechecker stores type information about each symbol in the global environment, which enables us to use the type of global functions during the code generation pass.

3.2 Code generation

3.2.1 Pattern-matching compilation

The pattern-matching compilation pass translates ML pattern-matching into a tree of Sqil's `case` selection instructions that is later translated to `switch` in C. The inherent backtracking involves the use of Sqil's lexical escape blocks, which generally compile to `goto`. This translation exactly follows the algorithm given in [Pey87], which is easy to implement but not optimal.

3.2.2 From ML abstract syntax tree to Sqil

We need two local environments for each toplevel phrase in this translation step:

1. An environment containing each variable with its access path from root to node in its defining pattern.
2. A local function environment containing the arity of each function name.

According to the different top nodes of the abstract syntax tree the translation proceeds in various ways:

- For the case of an identifier, we must distinguish a local or a global identifier, referring to a variable or to a function, because Sqil discriminates both kinds of values.
- For the case of a type constructor, we try to propagate constants, whenever it is possible thanks to the immutability of ML data types. Unfortunately this cannot be done with vectors and character strings that are inherently mutable in Caml-Light.
- The application case is the most complicated, because we want to uncurry most function calls:
  - When applying a local function or a primitive to a number of arguments matching its arity[2], we translate it to a direct application. But if some arguments are missing, we must add the code to build a partial function (using Sqil's `function` form); and if there are too many arguments, we have to compute additional calls using `funcall`.
  - The handling of global functions differs because the actual arity of an imported function is not accessible: only its type is known, and gives us its maximal arity. In order to increase execution speed, we insert several entry points for each global function, ranging from one argument to the maximum arity deducible from the function type. Thus a global function application simply calls the entry point corresponding to the number of available arguments.
  - If the functional value of the applicand is to be computed at runtime, we iterate `funcall` on each argument, since all functional values, i.e. closures, are of arity one in our scheme.
- The specialization of `let` needs the pattern-matching compilation pass and then distinguishes between variables, functions and recursive functions to be able to use `let`, `flet` and `labels` Sqil bivaluated forms.

[2]We define the *arity* as the number of arguments available when the evaluation of the function can begin.

- The CAML `try ... with` exception block is compiled by storing in a global variable the corresponding continuation, after having saved the old one. This permits us to raise an exception dynamically just by invoking the last created continuation.
- The translation of other constructs such as conditional forms or sequential forms is straightforward.

Finally we must remember to translate global definitions of functions and variables into `defun` and `defvar` Sqil forms.

3.2.3 The globalization pass

Up to this point we have not paid much attention to the fact that continuations and function values of Sqil are restricted to the toplevel environment. That ensures that these are simple adresses of either a stack frame or a global function entry.

If this is not the case, we have to globalize the function: a local function f that is referenced as a functional value must be moved to the toplevel and defined there with an additional argument representing its environment, i.e. its free variables. Then in general it is necessary to build a closure each time we need the functional value of f, and send its current environment together with its argument whenever f is called.

Moreover each free function of f will be either integrated in f if it is only referenced in f, or else must be globalized too. So any free variables of these functions must be present in f's environment too.

If a local function contains the form `(the-continuation)`, it must be globalized as well, and additional arguments must be supplied for its free variables.

3.2.4 C code expansion and linking

When compiling Sqil code to C, remember that such optimizations as elimination of tail-recursive calls, continuation displacement and integration of local functions are performed. Macros for data handling are expanded too.

The Caml-Light module system is simple but quite natural and is based on interface files that provide the names and types of all exported global identifiers of a module. It allows separate compilation between modules and the definition of abstract types such as polymorphic hashtables. So we have to link all compiled modules (along with the runtime support) to obtain the executable file.

The standard library is entirely written in ML; it provides C written primitives only for low-level tasks such as I/O or Unix system calls.

3.3 The runtime system

3.3.1 The selected memory model

All ML objects are uniformly represented by a machine word. Characters, integers, short reals and some special sum types such as booleans are immediate values whereas all other objects are statically or dynamically allocated in several words and represented by a pointer. We discuss below whether pointers and immediate values need to be distinguishable.

Closures and environments are simply allocated as a vector in consecutive memory words. So extending an environment by new variables is not implemented by chaining,

but needs to copy the values into a bigger environment vector. Sum types require a small integer tag for the purpose of discriminating different constructors.

3.3.2 Which is the best garbage collector?

In order to collect memory which is no longer used, we need a garbage collector that can trace ambiguous roots in the C execution stack. It is possible to use a *mark and sweep* algorithm in a BIBOP memory model without any tags in the objects themselves. With this scheme *all* fields in an allocated object are ambiguous.

We have tried this scheme and used Boehm's portable collector [BW88] for a while, but the results were not very good for programs that allocate a great amount of data in the stack or in the heap. The garbage collector was indeed spending too much time in its mark function because it considers every word in the stack or in the heap as a potential pointer.

We think it is much better to distinguish pointers and immediate values with a tag. First the difference between, say, an integer and a pointer in the heap and even in the stack is much more immediate. Then one can use a mostly copying algorithm, such as Delacour's one [Del91], which is more efficient because it does not scan the whole heap like a *mark and sweep* algorithm and because it increases locality by copying and compacting the structured objects.

Moreover the runtime system can easily offer the user 'generic' functions such as equal that need to explore the tree structure of the objects, and give these functions a polymorphic type.

Our implementation uses a one-bit tag scheme that renders all immediate values even and all pointers odd. This tag scheme gives better results than the opposite one, since arithmetic operations are little altered while pointer dereference only needs a small offset, which is free on many machines.

3.4 Benchmarks

Compiler	SML-NJ	Caml-Light	CeML	Zinc→K2	
Version	0.75	0.41		without GC	with GC
Sieve	4.2	11.6	2.4	1.1	1.2
Solitaire	21.3	124.6	7.4	4.7	4.7
Takeushi	18.7	49.6	3.5	6.1	6.1
Tak w. exceptions	33.0	73.6	77	45.2	45.8
Knuth-Bendix	1.9	8.8	12	2.7	3.1
Boyer	5.6	22.7	n.a.	3.9	4.1
Euclidian division	3.4	20.2	17.5	2.7	3.1
Church integers	5.5	31.3	25.2	3.8	3.8
List summation	24.8	41.4	10.6	6.1	8.3
Integral	34.1	59.7	14.2	5.5	13.3

Figure 1: Comparison among some ML compilers (user times in seconds)

Some benchmarks are listed on figure 1. They were performed on a Decstation 3100 with the R2000 RISC processor. Tests on a Sparc architecture yield similar results.

We have chosen three other good compilers to make some comparisons. These are the well-known SML-NJ native code compiler [AM87], the fast bytecode interpreter Caml-Light and another ML to C compiler, Emmanuel Chailloux's CeML [Cha91]. Roughly speaking, SML-NJ uses a Continuation Passing Style translation scheme with a fast *stop and copy* generational garbage collector, and allocates everything on the heap [App87]. Caml-Light uses a similar runtime as ours, but its collector is a non-conservative one with generations, whereas CeML has a good conservative *mark and sweep* algorithm, but scans a special-purpose application stack instead of the C stack and uses a type tag in the objects themselves.

The first tests – Erathostenes' Sieve, the game of Solitaire and Takeushi's function – show that the C generated code is at its best when compiling imperative programs. That was indeed expected! The next two – Tak using exceptions to return each partial result and the Knuth-Bendix completion algorithm – demonstrate that exception handling is quite expensive in C, because you generally have to use the library functions `setjmp` and `longjmp`.

Knuth-Bendix completion procedure, Boyer's tautology checker, Euclidian division (extracted from the Coq proof assistant) and Church integers are very functional programs. Zinc→K2's excellent performances with respect to these programs show that with our optimizing compilation strategy C code generation can achieve the best performance results for typical ML programs.

List summation tests list processing whereas Integral tests floating point arithmetic. Except for this last test, garbage collection is cheap (actually the version without a GC uses a simple allocation fringe pointer and objects have no tag). The reason for this overhead is that we cannot tag short reals, so we have to box them. An alternative would be to adopt a more clever boxing mechanism (see [Ler92]).

4 Conclusions

We have demonstrated that C code generation can be very efficient if one succeeds in making a good use of the control flow instructions present in C, despite its bad exception handling mechanism. Uncurrying is a critical factor in achieving this, because it eliminates a lot of redundant partial applications. The use of an appropriate intermediate language permits us to describe the relevant optimizations easily.

The next step would consist of giving an object its natural C type whenever it is known statically to have a monomorphic ML type. This would allow the C compiler to handle the object directly, yielding a better assembly code, and would avoid many useless heap allocations. We plan to develop this idea and to add pertinent datatypes to Sqil.

Finally, a garbage collector with ambiguous roots can be (nearly) as efficient as a conventional one, and a clever runtime tag mechanism does not hamper the overall performance.

5 Acknowledgements and related works

We would like to thank Xavier Leroy for his precious help, Emmanuel Chailloux for some valuable discussions, and Bruno Monsuez and Alan Mycroft respectively for their remarks about the first and second draft of this paper.

Previous work on the compilation of Lisp to C include the Kyoto Common Lisp [YH88] and Bartlett's SCHEME→C compiler [Bar89]. A previous attempt to compile ML to C is reported in [TAL90], but SML2C yields lower-level C programs, for example it does use an apply-like procedure for function calls instead of the standard C calling mechanism.

References

[AM87] A. Appel and D. MacQueen. A Standard ML compiler. In G. Kahn, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*. Springer-Verlag, September 1987. LNCS Vol. 274.

[App87] A. Appel. Garbage collection can be faster than stack allocation. *Informations Processing Letters*, June 1987.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bar89] J. F. Bartlett. SCHEME→C: a portable Scheme-to-C compiler. Technical report, Digital Equipment Corporation, January 1989.

[BW88] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[Cha91] E. Chailloux. Compilation des langages fonctionnels : CeML un traducteur ML vers C. Doctorat de l'Université Paris VII, November 1991.

[Cri91] R. Cridlig. Compilateur optimisant pour le langage ML. Technical report, École Polytechnique, Palaiseau, France, July 1991.

[Del91] V. Delacour. Gestion mémoire automatique pour langages de programmation de haut niveau. Doctorat de l'Université Paris VI, Juin 1991.

[IEE90] IEEE standard for the Scheme programming language. Technical Report 1178, IEEE Std, 1990.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Software Series. Prentice Hall, 1988.

[Ler90] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[Ler92] X. Leroy. Unboxed objects and polymorphic typing. *POPL*, 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice-Hall International, 1987.

[Sch88] W. Schelter. AKCL: Austin Kyoto Common Lisp. Unpublished(?), 1988.

[Sén89] N. Sénniak. Compilation de Scheme par spécialisation explicite. *Bigre*, 1(65), July 1989.

[Sén90] N. Séniak. Efficient compilation of local functions using C as a back-end. Technical report, LIX, École Polytechnique, Palaiseau, France, 1990.

[Sén91] N. Séniak. Théorie et pratique de Sqil, un langage intermédiaire pour la compilation des langages fonctionnels. Doctorat de l'Université Paris VI, October 1991.

[Ste84] G. L. Steele. *Common Lisp: the Language.* Digital Press, 1984.

[TAL90] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. Technical Report 187, CMU-CS, November 1990.

[YH88] T. Yuasa and M. Hagiya. Kyoto Common Lisp Report. Technical Report, Kyoto University, Research Institute for Mathematical Sciences, 1988.

An Efficient Way of Compiling ML to C

Emmanuel Chailloux
LIENS*- LITP†

Introduction

The ever increasing diffusion of ML, has yet to go beyond the areas of education and research. To change this situation, two conditions must be satisfied : in their shared characteristics, ML programs need to become as efficient as imperative programs without sacrificing security for efficiency and must also be easily interfaceable with other libraries. I propose a new compilation model for ML, called CeML [6], derived from the CAML [17] (ML dialect), which translates ML directly into C (understood as a portable assembler). In fact an abstract machine, inspired from FAM [5], is partially included in a runtime library. A particular attention has been paid to two points. The first one is function application which is optimized using an extended notion of functional type. The second one is the correspondence between basic ML types and basic C types. This allows ML objects including structured types, represented by pointers, to directly use the call protocol of C. This correspondence creates some difficulties for the Garbage Collector, but they vanish thanks to a new Mark&Sweep [12] with ambiguous roots [4], where basic values are not tagged.

All these implementation features are independent, making no supposition regarding the C compiler because they use another stack for the root set and for the application. In this manner, C programs created by CeML are easily interfaceable.

1 CeML Definition

CeML is a ML dialect derived from CAML language [17]. It differs from CAML with regard to its type system (with the introduction of `=>`, a new functional type constructor) and its module system.

The main syntactic restriction is the absence of local declaration for `type` and `exception`.

*URA 1327 - Laboratoire d'Informatique de l'École Normale Supérieure - 45 rue d'Ulm, 75230 Paris Cedex 05, France. Tel : (33)(1) 44.32.20.55 - Fax : (33)(1) 44.32.32.79 - Electronic mail: Emmanuel.Chailloux@ens.fr

†URA 248 - Laboratoire d'Informatique Théorique et Programmation - Institut Blaise Pascal - 4, place Jussieu - UPMC - 75252 Paris Cedex 05, France. Electronic Mail : ec@litp.ibp.fr

1.1 Type system

The λ-calculus has no notion of abstraction arity. The following notation : $\lambda xy.xy$ is just a convenient way of writing $\lambda x.\lambda y.xy$. But in a programming language we need to have this notion in order to obtain an efficient application. In CeML , we can define the function arity, bound to a function name, if there is no intermediate evaluation or local declaration between two (λ) (`function`). For example :

```
let addition = function x -> function y -> x+y;;
```

becomes a function with two arguments, and its type is `int => int => int` . This type is different from `int -> int -> int`, `->` which represents the classical functional type constructor. To formalize this type system, we give the following typing rules :

(App)

$$\frac{C \vdash expr_1 : \tau' \rightarrow \tau \quad C \vdash expr_2 : \tau'}{C \vdash expr_1 \quad expr_2 : \tau}$$

(TApp)

$$\frac{C \vdash expr : \tau_1 => \ \ldots \ => \tau_n => \tau \quad C \vdash expr_i : \tau_i (1 \leq i \leq n)}{C \vdash expr\ expr_1 \ldots expr_n : \tau}$$

(PApp)

$$\frac{C \vdash expr : \tau_1 => \ \ldots \ => \tau_n => \tau \quad C \vdash expr_i : \tau_i \quad (1 \leq i \leq k \ with \ k < n)}{C \vdash expr\ expr_1 \ldots expr_k : \tau_{k+1} \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau}$$

(E)

$$\frac{C \vdash \tau' => \tau}{C \vdash \tau' \rightarrow \tau}$$

(I)

$$\frac{C \vdash pat_i : \tau_i \quad expr : \tau}{C \vdash function\ pat_1 - > \ldots - > \ function\ pat_n - > expr : \tau_1 => \tau_2 => \ldots => \tau_n => \tau}$$

The great advantage of using a new type constructor to detect curried functions is the type propagation of the function result. The following function :

```
let add4 x y = let f u v = x+y+u+v in f;;
```

has this type : `int => int => (int => int => int)` which indicates the functional arity of the result.

remark In ML, user defined polymorphic function can explore the structure of a polymorphic parameter. In CeML, this rule is also applied to primitives. This is the reason why the symbol = has the semantic of **eq** (equality for immediate values or sharing for structured objects). It is a problem for the programmer but it is more coherent for the language definition because the ML polymorphism is parametric i.e. it does not look at the form of the function arguments.

For example, CeML includes a parametric polymorphism, = thus means **eq**, ie. describes an equality for immediate values or a sharing for structured values. Effectively = cannot mean **equal** because with parametric polymorphism a function does not explore the structure of a polymorphic parameter [14].

1.2 Module System

The CeML module system is not so powerful as CAML or SML module systems. It is only a convenient way to make separate compilation. Separate compilation is a convenient way of dividing programs into different compilation units. A CeML unit is decomposed into three parts : importation, implementation and exportation. The importation part describes the sub-graph of dependences at level one. The complete graph for a program has no cycles. The gray arrow of figure 1 is forbidden.

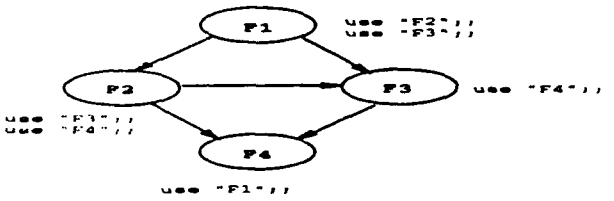


Figure 1: Dependencies graph

2 C as Intermediate Language

Many compilers consider C as a portable assembler. For example : some imperative languages (f2c , p2c), object languages (C++, Objective C, Eiffel, Modula3), logical languages (WAMCC) or functional languages (Scheme->C , KCL, SML2C , K2C , Z2K2C). These language references can be found in [6]. I present the main advantages and inconvenients to use the C language as assembler.

2.1 Advantages and inconveniences

2.1.1 advantages

- intrinsic : The memory allocation is sufficiently low level that it permits many kinds of manipulations. Global functions authorize many arguments, and control structures are powerful (loops, jump function for **switch**).
- standard : There is a standard ANSI C. The other C syntax (K&R [10]) is similar.

- efficiency : A great effort is effectuated by hardware manufacturers to obtain excellent C compilers on new processor architectures (RISC). One of the most popular benchmarks is the SPECMARK that gives a coefficient of a C compiler for a processor.
- tools : Many tools permit the profiling and debugging of the generated C code.

2.1.2 inconveniences

The main inconvenience in using C as an intermediate language is the absence of a good exception system. For example, C neither detects integer overflow nor C stack overflow.

2.2 What must be added to C to compile ML?

We need the following extensions to compile CeML to C :

- a Garbage Collector
- a general application mechanism
- an exception handling

These different extensions will be introduced into a runtime library that will need to be linked with C generated programs.

3 Compilation Scheme

The compilation scheme is classical, but its presentation gives a global vision.

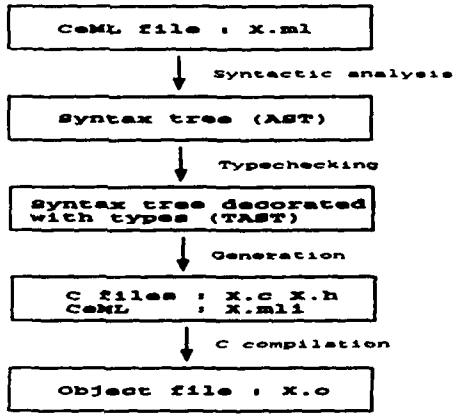


Figure 2: Compilation phases

For each CeML compilation unit (in fact a file with `.ml` extension), there is a corresponding C file (figure 2). After this compilation phase, the link phase (figure 3) creates an executable program from different units and from the runtime library .

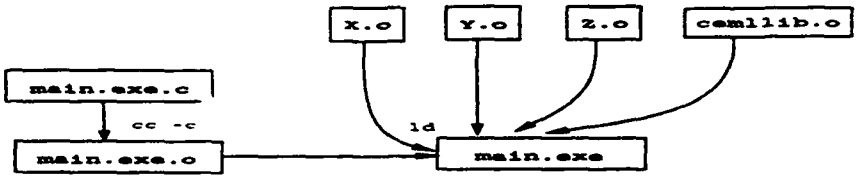


Figure 3: Link

CeML is actually a CAML program. For the computers supporting CAML it can be used directly as a command line. For the others, the C files must be generated on another computer and then be compiled on the host computer.

4 Runtime Library

In terms of efficiency, the runtime library is one of the essential parts of the implementation. I shall describe the type representation and a new Garbage Collector for the memory management, the application mechanism and the exception handling.

4.1 memory management

In ML, with a static typechecker, tags are effectively unnecessary [1]. In fact, only distinguishing information is needed for the summation types in order to distinguish constructors of a same type and the different kinds of vector. We follow this rule so as to be close to C basic types.

4.1.1 data type representation

Immediate values and pointers have a 32 bits representation. Others values (accessed via pointers) are represented in the heap by various structures. In order to distinguish between them, some type information is required.

Integers and floating point numbers use a word as well. Double precision floating point numbers are not implemented, but they can be represented as a pointer toward a four word storage.

4.1.2 Garbage Collector with ambiguous roots

I present a new `Mark&Sweep` algorithm with ambiguous roots. I describe the data representation, the partitioning of memory, the setroot and the algorithm used during the `Mark` phase to distinguish immediate values and pointers.

partitioning memory The heap is partitioned into chunks (cf. figure 5). Each chunk contains objects of the same size (to the power of two). There are *nbzones* sets of chunks called zone (from 2^3 to $2^{2+nbzones}$ bytes). Objects greater than one chunk are arranged into several chunks. This partitioning of memory is a variant of the **BIBOP**

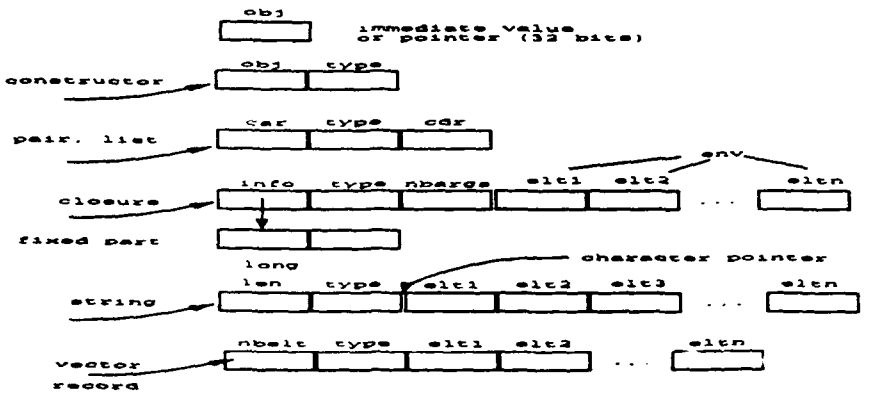


Figure 4: Data type representation

(Big Bag Of Pages [15]) algorithm. For our implementation, the chunk size is four kilobytes and *nbzones* is equal to ten.

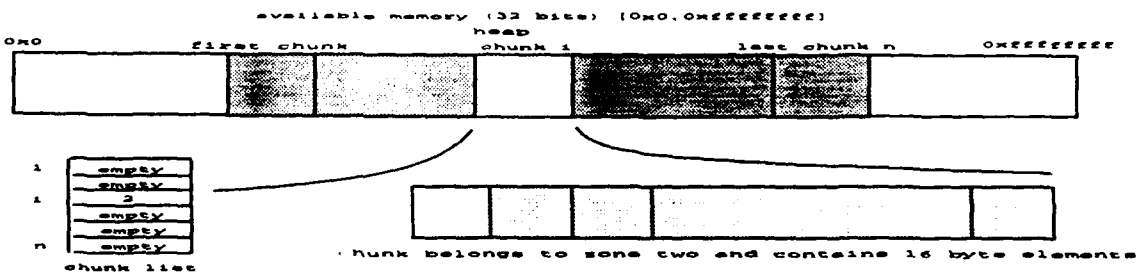


Figure 5: Partitioning memory

free lists Each predefined zone has a list of available elements (cf. figure 6).

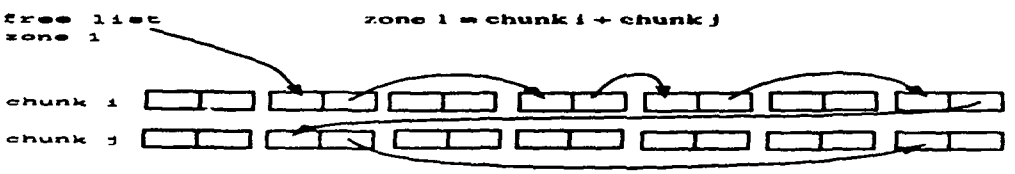


Figure 6: Free Lists

setroot The setroot is represented by a statically allocated independent stack. This stack allows for the storage of immediate values or pointers. Sometimes there is a double use between this stack and the C stack, but this choice is necessary so that the former stack may be independent of the C stack. This stack is also used for the general apply mechanism (when a direct call to a C function is not possible because the argument is a closure, or during a partial application).

initial memory state In the beginning, the heap is empty. Each zone can grow dynamically. The first allocation for a zone is ten chunks. The next allocations are computed by the growing function, after the **Sweep** phase. This orientation permits to control the heap evolution.

allocation There are two kinds of object allocation. The first one is used for small objects less than one chunk in size. This first case has two alternatives. When the object size is known (for example one *cons* uses four words) then, if the corresponding free list is not empty, the allocation is completed, otherwise a GC is invoked. In other cases the zone to be used has to be computed.
If the object size is greater than a chunk, then the object uses several contiguous chunks.

recovery When a zone is full, one must recover some space. There are two phases, the first (**Mark**) marks each object indicated by the setroot and the second (**Sweep**) preserves only these objects.

mark For each value inside the setroot, a discriminating algorithm distinguishes between an immediate value and a pointer. In this last case the structured object is marked and the process is applied to its structure elements. This algorithm is recursive, but it does not use the heap. Instead the recursive calls are pushed into the C stack.

sweep For each chunk in use, its corresponding free list is updated by all the unmarked elements. This algorithm explores all used memory. This is an implementation which wants to be simple. If the responsible zone which raised the GC is too small after memory recovery, then new chunks are allocated for this zone.

distinguishing algorithm When the GC examines a memory cell, it decides that its content should be considered as a pointer and marks the pointed object if the four following tests succeed :

- is the pointed object in the heap?
- does this address belong to a chunk in use?
- is the pointed object correctly aligned for this chunk?
- is there an object allocated to this address?

Otherwise, the content of the memory cell can be safely considered as an immediate value.

remark It is important to use a GC which displaces no objects, for the correspondence between CeML variables and C variables because otherwise the variable storage into the CeML stack is more costly (in this latter case two kinds of storage, one for C values and another for C variables are needed, and the pattern variables must also be stored in the root set).

4.2 General application mechanism

The general application mechanism uses the CeML stack, in order to push a closure and arguments. A closure is represented by a pair : (code,environment) where the code is a C function pointer and the environment a vector. More data is necessary in the closure, for example the maximum number of arguments (including environment values). The application expects a closure and i arguments on the top of the stack (cf. figure 7).

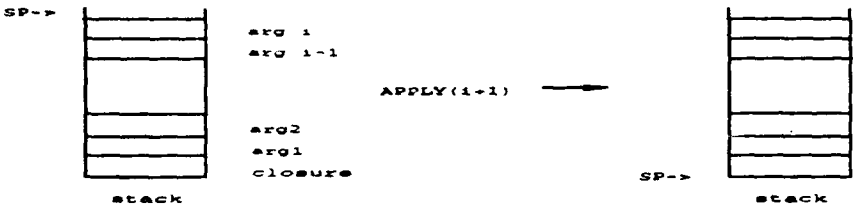


Figure 7: Application using stack

The return value of this application is considered as a C value, and returns to this evaluation step.

4.2.1 total application

The body evaluation of a function occurs when all its arguments have been passed. We note C_p^n a closure with n variables where p are already given. The total application begins if there are $i = (n - p)$ new arguments on the top of the stack. Let fn be the C function associated in the closure, the return value is:

$$fn(\overbrace{\underbrace{env[1], env[2], \ldots, env[p]}_{p}, \underbrace{Stack[SP+1], Stack[SP+2], \ldots, Stack[SP+i]}_{i}}^{n})$$

There is no closure duplication if all arguments are given.

4.2.2 partial application

In this case the environment portion of the closure is duplicated. The environment vector increases as in figure 8. For each partial application there is a closure duplication.

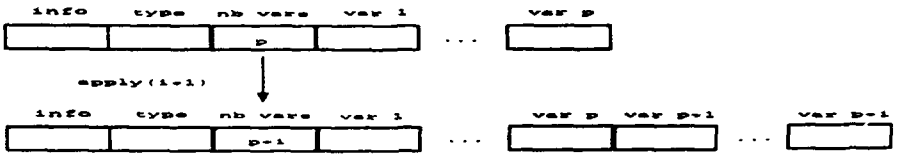


Figure 8: Closure duplication

4.3 Exceptions

CeML exceptions are considered as constructors, but this type is open.

The runtime library uses the C library `setjmp` and `longjmp`. `setjmp` is used for the `try` and `longjmp` for the `raise`. But it is necessary to add data to the C context (`jmp_buff`,) such as the CeML stack pointer. The return value from a `raise` is stored inside the stack. This stack pointer must be preserved in the CeML context. All the different CeML contexts are inside a linked list (EP is the head), as below :

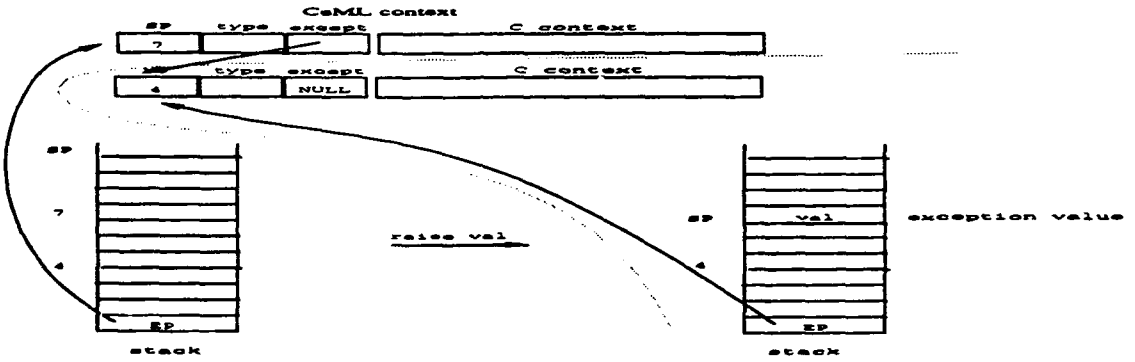


Figure 9: Raising an exception

5 Code Generator

I shall explain only the main features of declaration and expression generation.

There is a direct correspondence between CeML variables and C variables. During the generation phase, the CeML compiler uses four contexts, described below :

- stack context : is true if the expression must be pushed.
- variable context : has a variable name if the expression must modify this variable.
- return context : is true if the expression must be returned by a C function
- application context : is a list of composed applications.

I note that for $[\![expr]\!]_{stack,return,variable,applylist}$ or $[\![expr]\!]_{s,r,v,l}$ for the compilation of expressions in a compilation context. For the declarations, I write $[\![declvar]\!]_{(env,arity)}$ for a declaration with *env* as environment and *arity* as function arity. For example, a non functional variable has a zero arity, and global variables always have an empty environment. In all cases, a CeML variable is translated by one or two C variables.

5.1 non functional declarations

C authorizes local variables except for the functions, but closures have a C type defined as `Closure`. Then the translation for CeML variables of any type, except =>, is direct.

$$\begin{aligned}[\![let\ v = e]\!]_{([],0)} \quad = \quad & \{ typeC\ v\ ; \\ & [\![e]\!]_{true,false,"v=",[]}\ \}\end{aligned}$$

where *typeC* is the corresponding C type for the CeML type.

For the local and non functional declarations, the translation is identical because the local environment is known at this level.

5.2 Functional declarations

5.2.1 global declarations

In this case, the environment is empty. Then the number of parameters of the corresponding C function is the same as in CeML . In the following example, the f function has the type : $\alpha => \beta => \alpha$.

$$\begin{aligned}[\![let\ f = e]\!]_{([],2)} \quad = \quad & Closure\ \_M\_f\ ; \\ \\ & Ptr\ \_F\_f(\_V\_1, \_V\_2) \\ & Ptr\ \_V\_1\ : \\ & Ptr\ \_V\_2\ : \\ & push\_safe(\_V\_1)\ ; \\ & push(\_V\_2)\ : \\ & \{\ Ptr\ R\ ; \\ & [\![e]\!]_{false,false,"R=",[]} \\ & pop\_n(2)\ ; \\ & return(R); \}\end{aligned}$$

There are two C variables for each CeML function. One for the closure data structure and a second for the C function.

The previous example also initializes a C closure ($\_M\_f$) with its corresponding C function pointer ($\_F\_f$) and its arity (2), as follows :

$$\begin{aligned}[\![let\ f = e]\!]_{([],2)} \quad = \quad & \_M\_f\ =\ init\_closure("filename","f",\_M\_f,\_F\_f,2)\ : \\ & push(\_M\_f)\ ;\end{aligned}$$

5.2.2 local declaration

The binding of local function free variables is resolved by adding extra parameters to the C function, using λ-lifting [9]. Each free variable creates a new parameter, but it is not enough because a free functional variable can contain others free variables. The free variable set contains all variables not bound under a λ or a pattern-matching, and not defined at the global level. It is necessary to build a dependence variable set (DV) from the free variable set (FV) as follow :

$$DV_0(v_i) = FV(v_i)$$
$$(k \geq 0)\ if \quad DV_k(v_i) = w_j \ then\ DV_{k+1}(v_i) = \bigcup DV_k(F\{w_j\}) \cup V\{w_j\}$$

where $F\{w_j\}$ and $V\{w_j\}$ represent respectively the functional variables and the non-functional variables. of w_j, then $DV = DV_{k+1}$ if $DV_{k+1} = DV_k$.

The great difference between non recursive and recursive declarations comes from the initial free variable set.

non recursive declaration For the following CeML sentence (E) : $let\ v_1 = e_1\ and\ v_2 = e_2\ \ldots and\ v_n = e_n\ in\ e$ we obtain :

$$FV(v_i) = FV(e_i)$$
$$FV(E) = (\bigcup_{i=1}^{n} FV(v_i)) \cup (FV(e) - \{v_i\})$$

recursive declaration For the following CeML sentence (E) : $let\ rec\ v_1 = e_1\ and\ v_2 = e_2\ \ldots and\ v_n = e_n\ in\ e$ we obtain :

$$FV(v_i) = FV(e_i) - v_i$$
$$FV(E) = (\bigcup_{i=1}^{n} FV(e_i) \cup FV(e)) - \{v_i\}$$

translation Then the translation uses the dependence variable set :

$$\begin{aligned} [\![let\ rec\ v_1 = e_1\ and\ v_2 = e_2\ in\ e]\!]_{s,r,v,l} \ = \ & [\![let\ rec\ v_1 \ = \ e_1]\!]_{(DV(v_1),n1)} \\ & [\![let\ rec\ v_2 \ = \ e_2]\!]_{(DV(v_2),n2)} \\ & [\![e]\!]_{s,r,v,l}\} \end{aligned}$$

All the variables belonging to $DV(v_1)$ are translated as extra parameters to the corresponding C function.

5.3 Application optimization

The general case application is the following :

$$[\![expr_1\ expr_2]\!]_{(s,r,v,l)} = [\![expr_1]\!]_{true,false,"",[]}\ [\![expr_2]\!]_{true,false,"",[]}\ apply(2)_{s,r,v,l}$$

where *true* in stack position indicates that the expression must be pushed.

5.3.1 total application detection

The idea is to compare the function arity and the number of arguments given. If p arguments are given to f then :

- if f type is $t_1 -> t_2$ then it is the general case.
- if f type is $t_1 => t_2 => ... => t_{n+1}$ then there are two sub-cases :
  - $p < n$ then f followed by the p arguments are pushed and $apply(p+1)$ is called.
  - $p >= n$ then f is directly called with the first n arguments. If some arguments remain then another application is effectuated.

5.3.2 optimized application

Here we witness the use of C protocol call functions.

$$
\begin{aligned}
[\![f\ e_1\ \ldots\ e_n]\!]_{(s,r,v,l)} =_{opt}\ & \{typeC\ t_1\ T_1 1; \\
& \ldots \\
& typeC\ t_n\ T_n; \\
& [\![e_1]\!]_{false,false,"T1=",[]}\ push(T_1); \\
& \ldots \\
& [\![e_{n-1}]\!]_{false,false,"T_{n-1}",[]}\ push(T_{n-1}); \\
& [\![e_n]\!]_{"T_n="} \\
& pop\_n(n-1); \\
& \_F\_f(T_1, T_2, \ldots, T_n)(s,r,l,v)\ \}
\end{aligned}
$$

5.4 Pattern matching

For all types except integers and summation types, a naive algorithm (sequence of `if`) is used. For the integers and the summation types a **`switch`** control structure is used after a reorganization of the pattern matching. Constructors are represented by an integer. This optimization cannot be used with exceptions which belong to an open type.

6 Performances

I compare different ML compilers, divided into two families : CAML [17] and SML [13]. CAML and SML have a very similar core language. We can consider the following compilers compile the same core language. The CAML compiler translates to the CAM [7] code which is expanding into native code. CAML-LIGHT [11] translates to a byte code which is interpreted. SML/NJ [3] uses a CPS [2] model and produces native code. CeML and SML2C [16] generate C programs, but SML2C is the back end of SML/NJ and produces low level C programs, in contrast to that CeML generates high level C code.

All these benchmarks have effectively been run on the same machine (DecStation 3100 with a processor MIPS R2000) under the same conditions. All compilers, except CAML, create executable programs. CAML-LIGHT creates a byte code object but its loading is rapid. The `exportFn` function of SML/NJ produces a large executable program (a core image of approximately two megabytes). CeML and SML2C generate C programs that are passed on to the C compiler. The respective size of executable programs are reasonable (under one hundred kilobytes for small CeML programs, and about four hundred kilobytes for SML2C),

The following benchmarks try to test the differents parts of the ML language. Some of them are directly inspired from Gabriel's work [8]. `Fibonacci`, `Takeuchi`, `Integral` and `CountStr` use basic types as integers, floats and strings. `Takeuchi` is the curried form of the famous Takeuchi function. `Reverse`, `SigmaMap`, `ItList` and `Sieve` test the construction and access to lists inside functional programs. `Church int` and `DivEuclid` are very functional programs. The former carries out calculations on Church integers. The latter is the euclidian division extracted from the Coq system . `TakExcept` is always the Takeuchi function but written with exceptions. `KB` is the Knuth-Bendix term rewriting system applied to the group completion. It is very functional and uses exceptions. `SigmaVect` and `Soli_let` work on vectors. `Soli_let` is the resolution of the "solitaire game".

In figure 10, all the numbers represent user time on a Unix operating system (given in seconds). Bold numbers indicate the best results.

DS3100	CAML			SML		
Test	V2-6.1	light	CeML	NJ 0.66	SML2C	What is mainly tested?
Fibonacci	6.7	42.0	**2.5**	4.7	14.5	integers
Takeuchi	18.5	12.4	**0.7**	4.6	11.3	function calls (3 args)
Integral	4.0	6.7	**1.4**	**1.4**	3.8	floats
CountStr	12.5	**1.3**	1.9	6.6	10.3	strings
Reverse	14.6	9.6	**2.2**	2.6	6.4	list processing
SigmaMap	1.0	10.7	**0.6**	1.1	2.1	list processing, functionals
ItList	4.6	7.2	3.1	**2.1**	4.0	list processing, functionals
Sieve	7.5	13.2	**2.4**	4.0	10.7	list processing, functionals
Church int	5.4	10.4	6.4	**1.2**	4.8	functionals, polymorphism
DivEuclid	29.5	25.4	17.5	**3.8**	9.8	functionals, polymorphism
TakExcept	24.2	18.3	15.4	**7.2**	14.5	exceptions
KB	17.8	11.6	12	**2.6**	7.1	functionals, exceptions
SigmaVect	4.6	29.0	**1.3**	5.1	9.9	vectors, loops
Soli_let	25.4	151.0	**7.4**	29.0	*	vectors

Figure 10: Experimental results

CeML's excellent performance with respect to imperative programs was expected. But its results on the functional programs, when they are close to the Lisp style, justify, a posteriori, its implementation choices. Its less satisfying times on very functional programs are not really a hindrance because in general, a program is scarcely so functional.

Neverless, its performances on exceptions are really a problem if their use becomes a programming style.

Conclusion

In this paper, I have attempted to demonstrate that functional languages can obtain the same efficiency as imperative languages for their shared characteristics without sacrificing security for efficiency which would be a bad deal. This was globally verified. In many cases, the C program, generated by the CeML compiler, is very similar to the equivalent handwritten C program.

But at the same time, CeML had to be as efficient as the best ML implementations. This point is also globally verified. CeML performances compares very favorably with the best ML implementations. There are however two types of programs where CeML performances are not highly satisfactory : the very functional programs (close to λ-calculus as in the case of Church integers) and programs which use too much exception handling (as `Takeuchi` with exceptions). In the first case, the partial application and the application of closures given as arguments forbid the application optimization.

In the second case, the exception handling mechanism is above all dependent on the operating system and does not yield good results on an Unix system. But since the partial application and the intensive use of exceptions are in fact scarce, they do not put into question the CeML implementation choices.

In fact, with its more informative typechecker (arity of functions, expressions decorated with their types), the CeML compiler yields excellent optimizations for the total application and the manipulation of basic values. But theses optimizations are not always possible. Typically when the application depends on a functional argument, then its arity is lost for the application optimization. The execution speed then varies according to the ratio of non-optimized application / optimized application. Its GC, with ambiguous roots, avoids the tagging of immediate values. The distinguishing algorithm, between an immediate value and a pointer, slows the Garbage Collector down, but the benefits achieved through the uniform representation of data is, in most cases, greater than the slowdown.

One could also complain that the CeML stack replicate a part of the C stack. But for two reasons I think it is better to use an independent stack than only the C stack. First, it is necessary to be independent from the C implementation. The C stack is not specified in the C language definition. For this reason, we do not make any supposition about the C stack. Secondly, the content of the C stack is unknown and can be large. It is particularly important for the GC which scans this stack to mark objects. CeML loses time in building this stack but it can save time if many objects are inside the C stack. Although the GC needs to allocate chunks in good alignment, it accepts holes inside the heap, and nothing forbids mixed allocation between CeML and other C libraries.

C programs generated by CeML are readable. This property allows to use C tools to profile these programs and facilitates the debugging of the compiler.

The reduction in the difference of execution times between C and ML, and the interface possibilities with others libraries open the door for a more widespread use of functional languages as everyday programming languages.

References

[1] APPEL, A. Runtime Tags Aren't Necessary. *Lisp and Symbolic Computation* (1989).

[2] APPEL, A., AND JIM, T. Continuation-passing style, closure-passing style. *ACM on POPL* (1989).

[3] APPEL, A., MCQUEEN, D., AND DAVID, B. A standard ml compiler. *Functional Programming Languages and Computer Architecture* (1987).

[4] BOEHM, H., WEISER, M., AND BARTLETT, J. F. Garbage Collection in an Uncooperative Environment. *Software - Practice and Experience* (Sept. 1988).

[5] CARDELLI, L. The Functional Asbtract machine. *Polymorphism* (1983).

[6] CHAILLOUX, E. *Compilation des langages fonctionnels : CeML un traducteur ML vers C.* Thèse d'université, Université Paris VII, Nov. 1991.

[7] COUSINEAU, G., CURIEN, P. L., AND MAUNY, M. The Categorical Abstract Machine. *Functional Programming Languages and Computer Architecture* (1985).

[8] GABRIEL, R. P. *Performance and Evaluation of Lisp Systems.* Mit Press, Cambridge, Massachusetts, 1985.

[9] JOHNSSON, T. Lambda lifting: transforming programs to recursive equations. In *Conference on Functional Programing Languages and Computer Architecture. LNCS 201* (Nancy, 1985), ACM, Springer Verlag.

[10] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language.* Prentice-Hall, 1983.

[11] LEROY, X. The ZINC experiment : an economical implementation of the ML language. Tech. Rep. 117, INRIA, Feb. 1990.

[12] MCCARTHY, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM* (1960).

[13] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition Of Standard ML.* MIT Press, 1990.

[14] MORRISON, R., DEARLE, A., CONNOR, R. C. H., AND BROWN, L. An Ad Hoc Approach to the implementation of Polymorphism. In *Transaction on Programming Languages ans Systems* (1991), ACM.

[15] STEELE, G. L. Data Representation in PDP-10 Mac Lisp. In *MACSYMA Users Conference* (1977).

[16] TARDITI, D., AND ACHARYA, A. A guide to sml2c. Tech. rep., CMU-CS, June 1991.

[17] WEIS, P., APONTE, M. V., LAVILLE, A., MAUNY, M., AND SUAREZ, A. The CAML reference manual. Tech. Rep. 121, INRIA, Sept. 1990.

Standard ML for MS-Windows 3.0

Yngvi S. Guttesen
Department of Computer Science
The Technical University of Denmark
DK-2800 Lyngby

January 31, 1992

Abstract

Standard ML of New Jersey (SML_NJ) is a Standard ML compiler suitable for porting to different architectures. The compiler consists of a front end written i ML, a back end also written in ML, and a runtime system written i C and assembler. The front end is relatively independent of the target machine. This makes it easy to port the system to other architectures.

SML_NJ has up to now been reserved for workstations running UNIX. Code generators are available for a number of processors and the runtime system has been adapted to different UNIX versions.

The purpose of this paper is to report a new code generator for the Intel 80386 processor and a revised runtime system which runs under Windows. The code generator described in this paper is almost independent of the runtime system and consequently of the operating system, so it can be used in other architectures based on the 80386 processor, such as OS/2 and XENIX.

1 Introduction

Standard ML of New Jersey [4] is an implementation of the programming language Standard ML [6]. So far, Standard ML of New Jersey (SML_NJ) has solely been available on machines that run under the UNIX operation system. The purpose of this paper is to report a port of SML_NJ which runs on the Intel 80386 processor under Microsoft Windows 3.0, one of the most common combinations for PC's.

Our port of SML_NJ consists of two parts:

1. Standard ML modules which the SML_NJ code generator calls in order to produce 80386 machine code.
2. A new runtime system written in C and 80386 assembly language.

The first part is largely independent of Windows, i.e. it could be used for generating 80386 code that would run under other operating systems. The second part was necessitated by the fact that the runtime system has to run under Windows, which is very different from UNIX.

The reson we wrote parts of the runtime system in assembly language is that SML_NJ generates 32-bit code but no 32-bit C compiler is currently available under Windows.

When a 32-bit version of Windows becomes available, a new runtime system, which more closly resembles the original runtime system, can be written.

Few prerequisites are required to read this paper: familiarity with ML is neccesary and some knowledge about the Intel 80386 assembler is useful.

The rest of this paper is organized as follows: in Section 2 we review those aspects of the 80386/Windows platform that are of particular interest in this context. In Section 3 we report the new runtime system and in Section 4 we describe the new code generator. Finally in Section 5 we show an example.

Many of the technical details that are not covered in this paper can be found as comments in the source code. References are made to the relevant files, which are located in the *sml* distribution directory.

2 The 386/Windows platform

In this section we discuss those aspects of the 80386 and Windows, that are important in the context of SML_NJ. Windows causes troubles in implementing the system on a PC. This is because Windows still is based on the old 8086 memory model where the memory is divided into 64k segments. As explained in Section 2.2, this fits badly with the assumptions which SML_NJ makes about storage usage; these assumptions are described in Section 2.1. In Section 2.3 we describe how one can fit the two together.

2.1 Assumptions that SML_NJ relies on

The compiler operates with two basic datatypes — integers and pointers — both 32-bit[1]. Thus it is desirable that the architecture supports 32-bit data and addresses. The way memory is allocated and the way the garbage collection is done [1, 2] assumes that the ML code and data are located in a single continuous memory block. Since compiled ML code accesses variables and functions in the runtime system through 32-bit pointers and *vise versa*, the runtime system should be in the same logical memory space as the ML code/data.

The compiled ML code occasionally makes calls to the operating system, of which some are critical and other are of a more peripheral nature. That is, some of the system calls are used by the compiler itself whereas others just are for use in user applications. The critical system calls must be supported by the operating system, or at least it must be possible to simulate a corresponding action.

Languages like ML make heavy use of memory, so the operating system should provide some kind of virtual memory.

In the next two sections we will see how the 80386 and Windows meet these requirements. Only a brief overview together with the choices made will be given here. For a more elaborate discussion of the problems with choosing a memory model and how to build the system, the reader is referred to [5].

[1] actually only 31 because the least significant bit is used as a tag bit to distinguish between integers and pointers [2].

2.2 The 80386

The 80386 processor has 32-bit registers for manipulation of code and data. It is possible to address up to 2^{32} bytes (i.e. 4 Giga bytes), and the built-in paging mechanism makes virtual memory possible.

The memory organization is of interest to us. The 80386 operates with two kinds of segments:

> *USE16*: these are 16-bit segments. An address consists of a 16-bit segment address and a 16-bit offset. Code running in this type of segments by default uses 16-bit addresses and 16-bit data. Segment registers contain indices into a descriptor table that contains information about the type and location of the segment. With these segments the code and data has to be divided into 64K pieces, causing the well-known problems with managing large programs.
>
> *USE32*: these are 32-bit segments. An address consists of a 16-bit segment selector and a 32-bit offset. Code running in this type of segments by default uses 32-bit data and 32-bit addresses. This makes it possible to have a flat 32-bit memory model where the memory consists of a single contiguous block.

The same binary machine code will cause different actions in the two types of segments. When using USE16 segments, the 386 operates much like the predecessor 80286, only now 32-bit registers for manipulation of data are available. When using USE32 segments the processor acts like a real 32-bit processor.

The 80386 operates with separate code and data. It is not possible to write into a code segment nor to execute code from a data segment. But by letting a code segment register and a data segment register point to the same physical memory, self-modifying code becomes possible. This is known as *segment aliasing*.

The instruction set is comprehensive enough to implement the abstract machine that the front end generates code for. There are some problems in that the 386 has inherited many of the peculiarities from its predecessors. Some registers are dedicated to special purposes in some instructions and different addressing modes are available in different instructions. For details, see [5].

In short, the 80386 has what is needed to support the compiler. Unfortunately, Windows does not make all its capabilities available to the programmer.

2.3 Windows

Although Windows in enhanced mode exploits some of the processors 32-bit facilities, it continues to adhere the segmented memory model. It is *not* possible to implement a Windows application using an exclusively flat 32-bit memory model. Windows itself relies on 16-bit segments and a Windows application must contain at least one USE16 segment to interact with Windows.

Considerations about implementing the system using the USE16 memory model only, are given in [5]. Here I'll just state that it is not possible to get a satisfactory system within that model. Instead I'll explain how to run 32-bit programs under Windows.

The dynamic-link library WINMEM32.DLL which comes with the Software Development Kit (SDK) provides a set of functions that allow an application to make use of the 32-bit capabilities of the 80386 processor. It contains a function *Global32Alloc* that allocates a USE32 data segment (up to 16Mbytes) for which a code alias can be made with the function *Global32CodeAlias*, thus enabling segment aliasing. This makes it possible to generate and run 32-bit ML code within Windows.

As mentioned above (parts of) the runtime system should lie in the same address space as the ML code. We shall somehow move the runtime system into the ML-heap allocated with the *Global32Alloc* function. This can be done by collecting the relevant variables and functions of the runtime system in an assembler module, and then compiling and linking this module into a segment that can be copied into the ML-heap when initializing the system (the Microsoft C compiler cannot generate 32-bit code so we have to use the assembler). It is possible to access an USE32 segment from C code through a 64K "window" obtained by the *Global16PointerAlloc* function in the WINMEM32 library. This function allocates an USE16 alias to a portion (up to 64K) of the USE32 segment, but it is inconvenient and slow to use this technique when operating on larger portion of the ML heap. With the assembler one has completely control over and access to all the different kinds of segments.

There are three types of functions in the runtime system:

- functions that are called directly from the compiled ML-code must be placed in the ML-heap and therefore must be implemented in assembler.
- functions that interact closely with the ML-heap; these could be implemented in C by using USE16 aliases to access the ML heap, but have been implemented in assembler for greater efficiency.
- functions that hardly interact with the ML-heap; these can without any loss of efficiency be implemented in C.

The heart of the garbage collector is implemented in assembler together with some few utility functions (as are the functions that are called directly from ML). The rest of the runtime system is implemented in C. Access to the ML-heap is obtained through 64K "windows" allocated with *Global16PointerAlloc* function.

3 The runtime system

In this section the organization of the new runtime system is described in terms of the existing SML_NJ runtime systems.

The SML_NJ runtime system is written in C and assembler. It consists of:

- A carbage collector
- Small library functions written i assembler
- Small library functions written i C
- Facilities to handle Export and Import
- Facilities to handle signals (interrupts)

The runtime systems for the different UNIX versions are very alike. The differences are mostly expressed using #IFDEF OPSYS ... #ENDIF declarations in

the C-code. When writing a runtime system for Windows we cannot simply reuse the original C-code. As mentioned in the previous section, there are problems with mixing USE16 C-code and USE32 ML-code. However, I have tried to maintain the original structure of the runtime system, and only made modifications where needed because of the mixed memory model and the restrictions made by Windows. (Before reading the C code of the new runtime system, the reader is advised first to look at the original runtime system written for UNIX, because the many non-standard C details in the Windows version reduce the clarity.)

3.1 Segments

One of the first things the system does, when started, is to setup the USE32 segments that constitute the ML-heap. This is done by the code in the file *segments.c*. A USE32 segment (called *Use32Data*) is allocated and a code alias (called *Use32Code*) is established. These two constitute the ML-heap. The part of the runtime system that must lie on the ML-heap are placed in an assembler module (in the file *prim.asm*) which is compiled and linked into a segment (called *_RUNCODE*). The contents of this segment is copied into the ML-heap by the code that set up the USE32 segments.

3.2 The inteface between ML and the runtime system

After the system is initialized, control is passed to the compiled ML code. A C structure which contains the addresses of relevant variables and functions in the runtime system is passed on to the ML code. To call a C library function the ML code looks up its address *faddr* in a table and calls the *call_c* function with *faddr* as an argument.

Our interface must handle the context switch from USE16 to USE32 associated with the transfer of control from the runtime system (USE16) to the compiled ML code (USE32). When the ML code is entered, registers that the C compiler uses are saved on the stack. A C structure (*MLState*) holds the ML state. Before jumping to ML, registers are loaded from this structure. When returning to the runtime system things happens in reverse order.

When executing code in the runtime system the values in the segment registers are taken care of by the C compiler/linker and the Windows loader. Before jumping to the compiled ML-code we must load the segment registers with the correct values. DS is used as implicit segment register in most machine instructions. ES is used in string operations, and the SS segment register is used when ESP and EBP serve as index registers in memory references. DS, ES, and SS are loaded with the *Use32Data* value returned by *Global32Alloc*, before jumping to the ML-code. Care must be taken when switching stack in this way. When the USE16 stack is used, the upper 16 bits of the stack pointer (ESP) are not used, and can have random values. When switching to the USE32 stack we must ensure that the ESP register contains a legal stack address. After a move to the SS register the 80386 interrupt is disabled in one instruction, so we can handle this by the following code sequence:

```
...
mov ax, Use32Data
mov ss, ax                    ; no interrupts between
```

```
mov esp, Use32StackPointer      ; these two instructions
...
```

The context switch is handled by the function *restoreregs* in the file *interfce.asm* and by the functions *enterUse32* and *saveregs* in the file *prim.asm*, which also contains the variables and functions which must be placed in the ML-heap.

3.3 C library functions

The UNIX runtime system contains a number of C library functions that handle the interaction with the operating system. Some of these are UNIX specific and have no equivalent in Windows. In the Windows runtime system, these will generate a system exception if called:

```
void foo(ML_val_t arg)
{ raise_syserror("foo is not implemented under Windows!"); }
```

Certain functions that are not crucial to the compiler and that are difficult to implement under Windows are not implemented yet, and will also generate a system exception, if called.

The functions crucial to the compiler are all implemented. Some of these functions are UNIX specific and do not have an equivalent meaning in Windows, e.g. the masking and unmasking of signals. When called, those functions will perform some neutral action but will not cause a system exception. This will in some cases mean wrong return values to the ML system but these are not crucial. For example, Windows cannot perform all of the timer functions that are available in UNIX, so timing is not accurate in the Windows system. The C library functions are located in the file *callc.c*.

Details about the interaction between the runtime system and the ML code and data are found in [5], and in the comments in the files *prim.asm*, *gc.asm*, *interfce.asm*, and *util.asm*.

3.4 Assembly library functions

There are a number of assembler library functions that are called directly from compiled ML code. These are:

- *array(n,x)* allocates an array of length n, and initializes its elements to x.
- *callc(f,a)* calls the C function whose address is f with the argument a.
- *create_b(n)* allocates an uninitialized byte-array of length n.
- *create_s(n)* allocates a string of length n.
- *floor(x)* returns the "floor" of the real number x.
- *logb(x)* returns the exponent of the real number x.
- *scalb(x)* inserts a new exponent into the real number x.

Floating point operations are executed by using the 80387 math co-processor. We have added utility functions to handle the mixed memory model. Se comments in the file *util.asm*.

3.5 Garbage collection

The original garbage collector operates on the entire ML heap. Because it is difficult and inefficient to access USE32 data from the C code, We have implemented the heart of the garbage collector in assembler. It is actually just a "hand compilation" of the original garbage collector written in C, where 32-bit addresses and registers are used. The heart of the garbage collector is located in the file *gc.asm*, whereas other g.c. related functions are found in the file *callgc.c*.

3.6 Signal handling

In the UNIX version signal handlers have been implemented for a number of hardware signals. This is not possible in the Windows version. The whole signal machinery has been neutralized in the Windows version. The lack of signal handlers has some annoying consequences. For example it is not possible to stop an infinite loop with Ctrl-C.

As shown in [1] signals can, in a smooth way, be used to initiate garbage collection. All the existing implementations for UNIX use signals this way, either by allocating ahead until a pagefault occur, or by an explicit test for the available memory followed by an "interrupt on overflow". This can't be done in the Windows version. Instead we must make an explicit test and jump to a routine that can initiate the garbage collection when necessary.

```
                ...
                cmp    datalimit, allocptr     |
                jno    no_overflow             |  8 bytes
                jmp    initiate_gc             |
    no_overflow: ...
```

instead of:

```
                ...
                cmp    datalimit, allocptr     |
                into                           | 4 bytes
                ...
```

This is unpleasant because it takes up space (we shall perform this check at the beginning of every function).

3.7 Export and Import

In the UNIX version it is possible to export the state to an executable file (a.out format). This is used to make stand alone programs, and in particular it is used when bootstrapping the system to make "ready to run" versions of the interactive system and the batch compiler. Because of the problems that Windows has in handling USE32 segment this is not possible in Windows.

The ability to export the state to an executable file, has influence on the execution speed and the memory requirements. If exported to an executable file, the compiler code is located together with the runtime system below the base address of the ML heap, and will therefore not be collected when a major collection is performed [1]. This would in our system mean higher execution speed because most of

```
functor CPSgen( Machine : CMACHINE ) =
...
fun gen cexp =        (* generate code for the CPS-expression cexp *)
            ...
            Machine.subl3t(immed 1, regbind v, arithtemp)
            ... end
```

Figure 1: *CPSgen* converts CPS-expressions to machine code. Functions in the structure which CPSgen is applied to, shall generate the corresponding machine code, when called for by *CPSgen*.

the time is spent doing major collections. Because the heap always is at least three times the size of the living data [1, 2], the presence of the whole compiler on the heap contributes considerably to the memory requirements.

In principle, we could simulate the export by writing the contents of the ML-heap together with the contents of the runtime system's data segment to a file, and then use a special version of the runtime system to load and execute this file. This has not been done yet; instead the runtime system can load *.mo files as described in the *howto-boot* file in the SML_NJ documentation directory.

4 The code generator

The SML_NJ is nicely divided into a machine-independent front end and a machine-dependent back end. A single signature (called *CMACHINE*) defines the interface between the two. The back end consists of a few structures that — together with a few structures in the front end — are described below. At the end of this section some special aspects of the 80386 that has influence on the code generator are examined more closely. The intention with this section is to give a survey of the code generator and the description is therefore kept at a very high level.

4.1 From CPS-expressions to machine code

The front end transforms SML source code into CPS-expressions [3, 5, 7, 8]. The transformation of CPS-expressions into machine code is handled by the machine-independent functor *CPSgen*. *CPSgen* is parameterized on elementary code generating functions specified by the signature *CMACHINE*. That is, *CMACHINE* defines names for datatypes, variables, and functions that the front end will use when transforming CPS-expressions into machine code (see Figure 1). The *CPSgen* functor is located in the file *generic.sml*, and *CMACHINE* is found in *cmachine.sig*.

4.2 Backpatch and jumpsize-optimization

The back end has to handle relative addresses. That is, we have to backpatch relative jumps and other instructions that use relative addressing. To help in this, a machine-independent functor, *Backpatch*, is included in the compiler. *Backpatch* is

parameterized on a machine-dependent structure *Jump* which contains the information needed to backpatch on a particular machine. *Backpatch* needs to know the size in bytes of the instructions that use relative addressing and how to emit code for these. The signatures *BACKPATCH* and *JUMP* are given in Figure 2.

The machine code generator can make use of the primitives named in *BACKPATCH* (*emitstring, newlabel, etc*) when implementing the functions in *CMACHINE*. For example the *emitstring()* is used every time a string (code or data) is put into the code (see Figure 3). The full *Backpatch* functor, which is part of the general SML_NJ system, can be found in the file *backpatch.sml*. The *Jump* structure for the 80386 is found in the file *386jumps.sml*.

4.3 The 80386 code generator

As mentioned in section 4.1 we have to write a structure that matches the *CMACHINE* signature. A functor *CMach386* is used for this. The front end includes a batch compiler that can generate assembler code, so we need to generate both assembly code and machine code. The 'argument' given to *CMach386* determines which one is generated.

The signature *CODER386* defines a subset of the 80386 instruction set and addressing modes, that is used in the code generation. Two functors which match this signature

MCode386: generates machine code

ACode386: generates assembly code

are outlined in Figure 3.

We see how the functor *MCode386* is parameterized on the machine-dependent *Jump* structure mentioned above, and how this structure is passed on to *Backpatch*. This is how we get access to the primitives named in the *BACKPATCH* signature. When generating assembly code we do not need to backpatch and therefore the *Jump* structure is unnecessary. Instead symbolic labels are used. Notice that the primitives in *Backpatch* are used in the *MCode386* module only.

These two functors are used to make a 80386 code generator to be passed on to *CPSgen*. Applying the functor *CMach386* to one of them results in a structure whose functions when called, generates code (machine or assembly) for the 80386.

The signature *CODER386* is found in the file *386coder.sig*. The *Mcode386* and the *ACode386* functors are found in the files *386mcode.sml* and *386acode.sml*.

4.4 Putting it together

SML_NJ includes an interactive system and a batchcompiler. The front end functor *IntShare* defines the interactive system, and *Batch* defines the batchcompiler. Figure 4 shows how to build the interactive system and the batchcompiler for the 80386. *Batch* takes 2 arguments; *M* that generates machine code and *A* that generates assembly code. *Intshare* takes three argument where one of them (*Machm*) is the structure that generates the machine code. *Comp386* is the batchcompiler module and *Int386* is the interactive module.

```
signature BACKPATCH =                          (* Machine-independent          *)
sig
    eqtype Label
    type   JumpKind                            (* Note 1                       *)

    val newlabel  : unit           -> Label (* Create a new label             *)

    val define    : Label          -> unit  (* Associate a label with a point
                                               in the code                     *)
    val emitstring: string         -> unit  (* Insert a string into the code *)

    val align     : unit           -> unit  (* Ensure that the next code is
                                               on a 4-byte boundary            *)
    val jump      : JumpKind*Label -> unit  (* Insert a JumpKind instruction
                                               into the code (note 2)          *)
    val mark      : unit           -> unit  (* Insert a gc-tag in the code
                                               (note 3)                        *)
    val finish    : unit           -> string (* Initiate the backpatching.   *)
end

signature JUMPS =                                  (* Machine-dependent       *)
sig
  type JumpKind
  val sizejump : JumpKind*int*int*int -> int       (* return the size of the
                                                      JumpKind instruction     *)
  val emitjump : JumpKind*int*int*int -> string (* emit code for the JumpKind
                                                      instruction (note 1)     *)
  val emitlong : int -> string                     (* insert a 32-bit litteral
                                                      in the code              *)
end

functor Backpatch(Jump : JUMPS) : BACKPATCH =   (* Machine-independent       *)
struct
    open Jump
    ...
    datatype Desc = ... | JUMP of Jumpkind * Label * int ref * desc | ...
    ...
    fun jump(k,lab) = refs := JUMP(k,lab,ref 0, !refs)
    ...
end (* functor Backpatch *)
```

Note 1: The JumpKind datatype is used to encode the different instructions that uses relative addresses. The fragment of ***backpatch*** shown here indicates how the "jumps" are inserted into a tree to be backpatched later.
Note 2: The code generator calls this function when instructed to generate code for an instruction that refers to a relative address (i.e. to labels). *jump* makes a note of this instruction, and later when the backpatching in initiated, ***Backpatch*** can determine the size of the instruction (using *sizejump*) and how to emit code for it (using *emitjump*).
Note 3: Garbage collection (GC) is explained in [1].

Figure 2: The BACKPATCH and JUMP signatures

```
(* The basic machine code generation *)
functor MCode386(Jumps : JUMPS386) : CODER386 =
struct
    structure Emitter : BACKPATCH = Backpatch(Jumps)
    open Jumps Emitter
    ...
    datatype EA = ...
    ...
    fun movl(x : EA, y : EA) = emitstring ( ...  (* build up the string
                                                    that constitute a
                                                    move instruction *)
                                            ... )
        (* notice how the emitstring from Backpatch is used *)
    ...
end (* functor MCode386 *)

(* The basic assembly code generation *)
functor ACode386() : CODER386 =
struct
    ....
    datatype EA = ...
    ...
    fun movl(x : EA, y : EA) = (emit "mov   "; emit2args(x,y))
    ...
end (* functor ACode386 *)

(* The abstract machine *)
functor CMach386( Coder : CODER386) : CMACHINE =
    ...
    datatype EA = ...
    ...
    fun move(x : EA, y : EA) = Coder.movl(x,y)
    ...
end (* functor CMach386 *)
```

Note: When *CPSgen* calls the abstract instruction *move()* one of the *movl()* functions will be called to generate the desired code.

Figure 3: The basic code generators for the 80386 (machine and assembly code).

```
structure MC386 : CODEGENERATOR =
struct

  structure MachineCoder = MCode386(Jumps386)
  structure CMachine     = CMach386(MachineCoder)
  structure MachineGen   = CPScomp(CMachine)

  fun generate lexp = (MachineGen.compile lexp; MachineCoder.finish())

end (* structure MC386 *)

structure AC386 : ASSEMBLER =
struct

  structure AssemCoder = ACode386()
  structure CMachine   = CMach386(AssemCoder)
  structure AssemGen   = CPScomp(CMachine)

  fun generate(lexp,stream) = (Ass386.outfile := stream;
                               AssemGen.compile lexp)
end (* structure AC386 *)

structure Int386 = IntShare(structure Machm = MC386
                            val fileExtension = ".386"
                            structure D = BogusDbg
                           );

structure Comp386 = Batch(structure M=MC386 and A=AC386)
```

Figure 4: The 386 glue

4.5 Special conditions in 80386

The compiler was originally designed for the Vax and M680x0 family. Later on, code generators have been made for other architectures like MIPS and SPARC. The 80386 lacks some of their facilities and this has caused some problems.

One problem is the number of general registers. The *CMACHINE* signature specifies a set of variables. These are supposed to be in registers. But the 80386 has only seven general registers, so we have to simulate some extra registers in the memory. Where should these be placed? Under Windows it is possible to use absolute addresses because we manually allocate a segment to hold the ML code and data, and therefore have control over what goes where. Thus we can allocate say the $4n$ lowest addresses in the ML-heap to hold the n "memory registers". But in general (i.e. in other operating systems) we cannot use absolute addresses.

The runtime system (written in C) defines a set of variables that compiled ML code uses. These (or theirs addresses) are placed in a C-structure whose address is passed on to the ML code when initializing the system. In the same way we could allocate space in the runtime system (e.g. `int MemRegs[n]`) and pass *MemRegs* to the ML code. But this would require some changes in the front end, and the goal was to implement the code generator without any changes at all in the front end.

As a result of Continuation Passing Style, ML does not use the stack. We therefore have a static stack when running ML code, so the "memory registers" can be put on the stack in the same way as one would allocate space for local variables on the stack in a C-compiler. This is what has been done here. But care must be taken when referring to these variables. Some registers in the 80386 are dedicated to special purposes in some instructions. Consequently, the stack is used by a few functions in the code generator to save the value of these registers, when necessary. But this is not a problem because we always know exactly how much has been pushed onto the stack.

Using the stack to hold the memory registers has another advantage over absolute addresses in that we save 2 bytes in the code size on every reference to memory variables. The code size is a serious problem when running under Windows because we are restricted to 16 Mbyte.

Because the 80386 cannot perform memory to memory operations, a lot of moving to temporary registers takes place, which again contributes to the code size. This is a serious problem with the 80386 when running under Windows.

Another problem with the Intel 80386 microprocessor is that it does not directly support PC-relative addressing. SML_NJ provides facilities to handle this problem. The front end keeps track of the addressing requirements of each function and calls the *beginStdFn* in *CMACHINE* if it uses PC-relative addressing. One of the parameters to *beginStdFn* is the closure of the function so *beginStdFn* can load the base address and later use it for relative addressing (see the file *cmachine.sig* for details). But in version 0.66 of the compiler, the *needPC* function (located in *CPSgen*) that determine whether a function uses PC-relative addressing is constantly *true*:

```
fun needPC cexp = true
```

Thus, if the above technique is used, every function gets the extra overhead with calculating a function's base address. Hoping that many functions don't use PC-relative addressing I have chosen another technique. We need to generate code for instructions like *load effective address*:

```
LEA reg1, (PC,offs)     ; reg1 := PC+offs
```

This can be done by the following piece of code:

```
1000: call  0           ; 1005 pushed (relative call)
1005: pop   eax         ; eax = 1005
1006: sub   eax, offs+5 ; +5 because the size of the call
                        ; instruction is 5 bytes
```

This way we can simulate PC-relative addressing. The size of these three instructions is 12 bytes, so if this kind of addressing is performed often it is better to use *beginStdFn* as described above. The frequency of PC-relative addressing ought to be measured to determine which technique is better.

This concludes the description of the 80386 code generator.

5 Example

In this sections we show how a simple function is compiled with the 80386 code generator. The assembly code shown below is generated with the assemble command in the batch compiler (see the file *BATCHINSTALL*).

```
fun f x = if x=0 then 1 else x*f(x-1)
```

is compiled into:

```
; [esp+00] is the datalimit ''register'' and holds the highest available addr.
; [esp+12] holds the address of the rutine to initiate g.c.
: [eps+20] holds the address of the rutine to handle overflow
; [esp+40] is the standard argument ''register''
; [esp+44] is the standard continuation ''register''
; [esp+48] is the standard closure ''register''
  ...

L3:
cmp   dword ptr [esp+0],edi     ; check the available memory
jns @f
call    dword ptr [esp+12]      ; and call garbage collection if necessary
@@:
cmp   dword ptr [esp+40],1      ; if x=0
jne   L14
mov   edx,dword ptr [ebp+0]     ; then continue with 1
mov   dword ptr [esp+44],ebp
mov   dword ptr [esp+40],3
jmp   edx
L14:                            ; else x*f(x-1)
mov   eax,49
stos  eax                       ; setup a closure with the argument x and
lea   eax,L4                    ; the continuation for the multiplication L4
stos  eax
mov   eax,dword ptr [esp+40]
```

```
stos  eax
mov   eax,ebp
stos  eax
lea   ebp,dword ptr [edi+-12]
sub   dword ptr [esp+40],2      ; x=x-1
jno @f                          ; check for overflow
call    dword ptr [esp+20]
@@:
jmp   L3                        ; make the call f(x-1)

...

L4:
cmp   dword ptr [esp+0],edi     ; check the available memory
jns @f
call    dword ptr [esp+12]      ; and call G.C. if neseccary
@@:
mov   eax,dword ptr [esp+44]    ; make the multiplications
mov   eax,dword ptr [eax+4]
mov   dword ptr [esp+48],eax
mov   ebx,dword ptr [esp+48]
sar   ebx,1
sub   dword ptr [esp+40],1
mov   ecx,dword ptr [esp+40]
imul  ecx,ebx
mov   dword ptr [esp+40],ecx
jno @f
call    dword ptr [esp+20]
@@:
add   dword ptr [esp+40],1
mov   eax,dword ptr [esp+44]
mov   eax,dword ptr [eax+8]
mov   dword ptr [esp+44],eax
mov   eax,dword ptr [esp+44]
mov   edx,dword ptr [eax+0]
jmp   edx                       ; and continue
...
```

Notice how the stack is used to simulate registers, and how the EAX and ECX are used as tempory registers. In SML_NJ an integer i is represented as $i*2+1$, so the integer one (1) is represented as the integer three (3).

6 Conclusion

We have succeded in porting the SML_NJ version 0.66 to a PC running Windows. A general 80386 code generator is made, which can be used in other architectures based on the 80386 processor. The new runtime system lacks some of the facilities found in the UNIX version, but we will not continue working on the runtime system until new 32-bit versions of Windows and C compilers becomes available. The purpose of this project was to see if it was possible to implement the whole SML_NJ system on a PC, and that we have proved.

No changes have been made in the front end[2], so the whole system is implemented. Other compilers for a subset of ML are available on the PC, but this is the first time (to our knowledge) that a complete SML compiler runs on the PC.

References

[1] Andrew W. Appel. "Simpel generational garbage collection and fast allocation". Software Practice and Experience, Feb. 1989.

[2] Andrew W. Appel. "A Runtime System", LISP AND SYMBOLIC COMPUTATION: An International Journal, 3, 343-380, 1990. Kluwer Academic Publishers, Manufactured in The Netherlands.

[3] Andrew W. Appel. "Continuation-Passing, Closure-Passing Style", Sixteenth ACM Symp. on Principles of Programming Languages 293-302, 1989.

[4] Andrew W. Appel and David B. MacQueen. "A Standard ML compiler". Functional Programming Languages and Computer Architecture (LNCS 274), pp. 301-324. Springer-Verlag, 1987.

[5] Yngvi S. Guttesen. "Flytning af SML til en PC", Master Thesis, ID-E-542, 1991. Department of Computer Science, The Technical University of Denmark.

[6] R. Milner, M. Tofte, and R. Harper. "The Definition of Standard ML", MIT Press, 1990.

[7] D. Kranz, R. Kelsey, J. Rees, P. Hudsk, J. Philbin, N. Adams. "Orbit: An optimizing compiler for Scheeme". Proc. Sigplan '86 Symp. on Compiler Construction, vol. 21 (Sigplan Notices), no. 7, pp. 219-233, July 1986.

[8] Guy L. Steele. "Rabbit: a compiler for Scheeme". AI-TR-474, MIT, 1978.

[2] We had to change some structure names because filenames in DOS can only be upto 8 characters in length.

Completely Bounded Quantification is Decidable

Dinesh Katiyar Sriram Sankar*
Stanford University
California, USA

Abstract

This paper proves the decidability of subtyping for $F_{\leq}$ when the bounds on polymorphic types do not contain *Top* (*i.e.*, in all types of the form $\forall\alpha<:\tau_1.\tau_2$, τ_1 does not contain *Top*). This general restriction is subsequently relaxed to allow unbounded quantification.

1 Introduction

$F_{\leq}$ [CW85,CG] is a typed λ-calculus with subtyping and bounded second-order polymorphism. The importance of $F_{\leq}$ in programming language design is that it provides a simple context for studying the typing problems that arise when subtyping and bounded quantification are added to polymorphic languages such as ML.

Curien and Ghelli [CG] recently developed a subtyping algorithm for $F_{\leq}$ and proved its partial correctness. Subsequently, Ghelli [Ghe90] presented a termination proof for this algorithm. A mistake was discovered in this termination proof, following which Pierce [Pie92] presented a proof showing that the subtyping problem for general $F_{\leq}$ types is undecidable.

This paper shows how one can make the subtyping problem decidable by imposing some restrictions on $F_{\leq}$ types. We first prove the termination of Curien and Ghelli's algorithm when the bounds on all polymorphic types involved do not contain *Top*. *i.e.*, In all types of the form $\forall\alpha<:\tau_1.\tau_2$, τ_1 does not contain *Top*. Such a bound completely determines the structure of α, hence we refer to this as "completely bounded". We later show that this restriction can be relaxed to allow unbounded quantification. Adding records and unions to our system causes it to become undecidable. We are currently working on defining subset restrictions for records and unions similar to those presented in this paper to make subtype checking decidable in the presence of these types.

We are in the process of designing a type system based on $F_{\leq}$ for a prototyping language called *Rapide* [BL90,MMM91], and are implementing a subtyping algorithm for this type system. Given the undecidability of subtyping for general $F_{\leq}$ types, we need to restrict our type system so as to

*ERL449, Computer Systems Laboratory, Stanford University, Stanford, California - 94305. *phone:* (415)723-1835. *email:* sankar@cs.stanford.edu.

make subtyping decidable. Results such as those presented in this paper will aid in determining the necessary restrictions.

In Section 2, we present Curien and Ghelli's algorithm. We prove the termination of this algorithm for completely bounded quantification in Section 3. Section 4 shows how we can relax our restrictions to allow unbounded quantification. Section 5 presents examples of records and unions that cannot be handled by simple extensions of Curien and Ghelli's algorithm. Section 6 concludes the paper by describing plans for future work.

2 Curien and Ghelli's algorithm

A $F_{\leq}$ type τ is either a simple type (such as *Int*), a variable, a function type $(\tau_1 \rightarrow \tau_2)$, or a polymorphic type $(\forall \alpha <: \tau_1 . \tau_2)$. Given a list of assumptions Γ and two types σ and τ, the subtyping problem is to determine whether or not $\Gamma \vdash \sigma <: \tau$ — *i.e.*, whether or not σ is a subtype of τ given the assumptions in Γ. The assumptions in Γ will all be of the form $\alpha <: \tau$, where α is a type variable and τ is a type. (This convention — that α's refer to type variables and σ's and τ's refer to types — is used for the rest of the paper. We shall also use α to refer to simple types — the context will make it clear whether a particular α is a variable or a simple type.) τ in this case is called the *bound* of α and is referred to as $\Gamma(\alpha)$. Free variables in τ may only be *bounded* in other assumptions in Γ to the left (earlier in the list) of the assumption containing τ.

Curien and Ghelli's algorithm is presented as a list of axiom schemas and inference rules. These schemas and rules contain templates of subtyping problems. The algorithm proceeds by applying the inference rules backwards to the subtyping problem. If the subtyping problem matches the template below the line of an inference rule, it reduces to subtyping problems that can be derived from the templates above the line of the inference rule. If the subtyping problem matches the template of an axiom schema, the algorithm reports success. In all other cases, the algorithm reports failure.

The axiom schemas and inference rules are listed below:

(NTop) $$\Gamma \vdash \sigma <: \mathit{Top}$$

(NRefl) $$\Gamma \vdash \alpha <: \alpha$$

(NVar) $$\frac{\Gamma \vdash \Gamma(\alpha) <: \tau}{\Gamma \vdash \alpha <: \tau}$$

(NArrow) $$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \Gamma \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2 <: \tau_1 \rightarrow \tau_2}$$

(NAll) $$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \Gamma, \alpha <: \tau_1 \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \forall \alpha <: \sigma_1 . \sigma_2 <: \forall \alpha <: \tau_1 . \tau_2}$$

3 Proof of termination

We prove termination by defining a complexity metric which is finite and positive for each subtyping problem, and show that the application of inference rules causes the complexity of the new subtyping problems generated to decrease. Since the complexity cannot decrease indefinitely, the algorithm will have to terminate.

In 3.1, we define the complexity metric. In 3.2, we show how the complexity metric is affected by applying the various inference rules. We conclude with a condition (Theorem 1) that, if satisfied, will guarantee the termination of the subtyping algorithm. Finally, in 3.3, we show how our restrictions on bounds of polymorphic types satisfies the condition of Theorem 1.

3.1 The complexity metric

We define the *size* of a type τ with respect to a list of assumptions Γ, and refer to this as $size(\tau)_\Gamma$. The *complexity* of a subtyping problem $\Gamma \vdash \sigma <: \tau$ is defined as:

$$complexity(\Gamma \vdash \sigma <: \tau) \stackrel{\text{def}}{=} size(\sigma)_\Gamma + size(\tau)_\Gamma$$

$size(\tau)_\Gamma$ is determined by recursively replacing variables in τ with their respective bounds and then computing the textual size of the resulting expression. $size(\tau)_\Gamma$ is formally defined as:

$$\begin{aligned}
size(Top)_\Gamma &= 1 \\
size(\alpha)_\Gamma &= \begin{cases} size(\Gamma(\alpha))_\Gamma & \text{if } \Gamma(\alpha) \text{ is defined} \\ 1 & \text{otherwise} \end{cases} \\
size(\tau_1 \rightarrow \tau_2)_\Gamma &= size(\tau_1)_\Gamma + size(\tau_2)_\Gamma \\
size(\forall \alpha <: \tau_1 . \tau_2)_\Gamma &= size(\tau_1)_\Gamma + size(\tau_2)_{\Gamma, \alpha <: \tau_1}
\end{aligned}$$

Examples:

1. $size(Top \rightarrow \alpha)_{\alpha <: Int \rightarrow Int}$
   $= size(Top)_{\alpha <: Int \rightarrow Int} + size(\alpha)_{\alpha <: Int \rightarrow Int}$
   $= 1 + size(Int \rightarrow Int)_{\alpha <: Int \rightarrow Int}$
   $= 1 + size(Int)_{\alpha <: Int \rightarrow Int} + size(Int)_{\alpha <: Int \rightarrow Int}$
   $= 1 + 1 + 1 = 3.$

2. $size(\forall \alpha_1 <: (\alpha_2 \rightarrow Int).\alpha_1)_{\alpha_2 <: Int \rightarrow Int}$
   $= size(\alpha_2 \rightarrow Int)_{\alpha_2 <: Int \rightarrow Int} + size(\alpha_1)_{\alpha_2 <: Int \rightarrow Int, \alpha_1 <: \alpha_2 \rightarrow Int}$
   $= size(\alpha_2)_{\alpha_2 <: Int \rightarrow Int} + size(Int)_{\ldots} + size(\alpha_2 \rightarrow Int)_{\alpha_2 <: Int \rightarrow Int, \alpha_1 <: \alpha_2 \rightarrow Int}$
   $= size(Int \rightarrow Int)_{\ldots} + 1 + size(\alpha_2)_{\alpha_2 <: Int \rightarrow Int, \alpha_1 <: \alpha_2 \rightarrow Int} + size(Int)_{\ldots}$
   $= 2 + 1 + size(Int \rightarrow Int)_{\ldots} + 1$
   $= 2 + 1 + 2 + 1 = 6.$

Lemma 1 *$size(\tau)_\Gamma$ is always finite and positive.*

Proof. It is obvious from the definition of *size* that it has to be positive. We prove that it is finite by showing that evaluation of *size* terminates for all τ and Γ. A complexity metric similar to that used in Ghelli's flawed termination proof actually works in this case.

The complexity metric to prove the termination of the evaluation of $size(\tau)_\Gamma$ is obtained by first ordering all the variables that occur in τ and Γ such that the following property is satisfied: If α_i is defined in the bound of α_j, then α_i occurs to the left of α_j in the ordering. It is possible to obtain such an ordering given the structure of $F_{\leq}$ (there may be multiple orderings that satisfy this condition in which case, one of them is chosen arbitrarily). The *depth* of each variable is then defined as the number of variables that occur to the left of it in this ordering.

The complexity of any subproblem $size(\tau')_{\Gamma'}$ that arises during the evaluation of $size(\tau)_\Gamma$ is the tuple $\langle D, S\rangle$, where D is the maximum depth over all the variables that occur in τ' and S is the textual length of τ'. There may be variables in τ' and Γ' that do not occur in either of τ or Γ. These variables are created when an existing variable is duplicated in the reduction process, thus requiring one of the uses to be renamed. The depth of the renamed variable is defined to be the same as its unrenamed counterpart. The key to this proof is that defining the depth of renamed variables in this manner maintains the condition based on which the initial ordering was created.

It is easy to see that the complexity decreases (the ordering between $\langle D, S\rangle$ tuples is lexicographic) during the evaluation of $size(\tau)_\Gamma$. For all reductions other than $size(\alpha)_\Gamma = size(\Gamma(\alpha))_\Gamma$, the D component of the complexity metric either remains the same or decreases while the S component decreases; whereas in the abovementioned reduction, the D component decreases.

Since this complexity metric cannot decrease indefinitely, the evaluation of *size* for any τ and Γ has to terminate. □

3.2 The effect of inference rule application on complexity

NVAR

$$complexity(\Gamma \vdash \alpha <: \tau) = size(\alpha)_\Gamma + size(\tau)_\Gamma = size(\Gamma(\alpha))_\Gamma + size(\tau)_\Gamma = complexity(\Gamma \vdash \Gamma(\alpha) <: \tau)$$

i.e., The complexity metric remains the same after application of the inference rule NVAR. However, NVAR may be applied continuously at most as many times as there are variables in Γ before one of the other rules has to be applied. The complexity metric reduces when any of the other rules are applied, so there is no problem.

NARROW

It is quite obvious that $complexity(\Gamma \vdash \tau_1 <: \sigma_1)$ and $complexity(\Gamma \vdash \sigma_2 <: \tau_2)$ are both less than $complexity(\Gamma \vdash \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2)$.

NALL

It is quite obvious that $complexity(\Gamma \vdash \tau_1 <: \sigma_1)$ is less than $complexity(\Gamma \vdash \forall\alpha<:\sigma_1.\sigma_2 <: \forall\alpha<:\tau_1.\tau_2)$. We shall now simplify $complexity(\Gamma \vdash \forall\alpha<:\sigma_1.\sigma_2 <: \forall\alpha<:\tau_1.\tau_2) - complexity(\Gamma, \alpha<:\tau_1 \vdash \sigma_2 <: \tau_2)$ to determine the conditions under which the complexity metric decreases when reducing to the second rule above the line in NALL. This expression, which must be positive for the complexity metric to decrease, simplifies as follows:

$$
\begin{aligned}
&complexity(\Gamma \vdash \forall\alpha<:\sigma_1.\sigma_2 <: \forall\alpha<:\tau_1.\tau_2) - complexity(\Gamma, \alpha<:\tau_1 \vdash \sigma_2 <: \tau_2) \\
&= (size(\forall\alpha <: \sigma_1.\sigma_2)_\Gamma + size(\forall\alpha <: \tau_1.\tau_2)_\Gamma) - (size(\sigma_2)_{\Gamma,\alpha<:\tau_1} + size(\tau_2)_{\Gamma,\alpha<:\tau_1}) \\
&= (size(\sigma_1)_\Gamma + size(\sigma_2)_{\Gamma,\alpha<:\sigma_1} + size(\tau_1)_\Gamma + size(\tau_2)_{\Gamma,\alpha<:\tau_1}) - (size(\sigma_2)_{\Gamma,\alpha<:\tau_1} + size(\tau_2)_{\Gamma,\alpha<:\tau_1}) \\
&= size(\sigma_1)_\Gamma + size(\tau_1)_\Gamma + (size(\sigma_2)_{\Gamma,\alpha<:\sigma_1} - size(\sigma_2)_{\Gamma,\alpha<:\tau_1})
\end{aligned}
$$

We concentrate on the part $size(\sigma_2)_{\Gamma,\alpha<:\sigma_1} - size(\sigma_2)_{\Gamma,\alpha<:\tau_1}$ from the last line above. If this expression is non-negative, then the overall expression will be positive, and therefore the complexity metric will decrease on the application of NALL. We need to present the following lemma before we can proceed further.

Lemma 2 *For any types σ, τ, and for any list of assumptions Γ, such that the variable α is not bounded in any of them, and also does not occur anywhere in Γ, $size(\sigma)_{\Gamma,\alpha<:\tau} = size(\sigma)_\Gamma + n(size(\tau)_\Gamma - 1)$, where $n \geq 0$ and depends only on σ.*

The evaluation of $size(\sigma)_{\Gamma'}$ (for any Γ') will reduce to zero or more evaluations of the form $size(\alpha)_{\Gamma''}$ in addition to other reductions. Reductions to the form $size(\alpha)_{\Gamma''}$ may be either due to occurrences of α in σ, or occurrences of α in bounds in Γ' which are used to replace the variables they bound. If Γ' does not contain any bound that contains α, then the number of times $size(\sigma)_{\Gamma'}$ reduces to the form $size(\alpha)_{\Gamma''}$ depends only on σ. Suppose this number is n.

Therefore, $size(\sigma)_\Gamma$ reduces to n evaluations of the form $size(\alpha)_{\Gamma''}$, each of which evaluates to 1 since α is not bounded anywhere. So we can write $size(\sigma)_\Gamma$ as $m + n$ where m is the result of the evaluation of the remainder of the reductions.

Similarly, $size(\sigma)_{\Gamma,\alpha<:\tau}$ reduces to n evaluations of the form $size(\alpha)_{\Gamma''}$, while the remainder of the reductions evaluate to m. Each reduction to $size(\alpha)_{\Gamma''}$ further reduces to $size(\tau)_{\Gamma''}$. Since Γ'' is of the form $\Gamma, \alpha<:\tau, \ldots,$ τ does not depend on the portion of Γ'' to the right of Γ. Therefore, $size(\tau)_{\Gamma''} = size(\tau)_\Gamma$. So we can write $size(\sigma)_{\Gamma,\alpha<:\tau}$ as $m + n(size(\tau)_\Gamma)$, which is the same as $size(\sigma)_\Gamma + n(size(\tau)_\Gamma - 1)$. □

We now simplify $size(\sigma_2)_{\Gamma,\alpha<:\sigma_1} - size(\sigma_2)_{\Gamma,\alpha<:\tau_1}$.

$$
\begin{aligned}
&size(\sigma_2)_{\Gamma,\alpha<:\sigma_1} - size(\sigma_2)_{\Gamma,\alpha<:\tau_1} \\
&= (size(\sigma_2)_\Gamma + n(size(\sigma_1)_\Gamma - 1)) - (size(\sigma_2)_\Gamma + n(size(\tau_1)_\Gamma - 1)) \qquad \text{(for some } n \geq 0) \\
&= n(size(\sigma_1)_\Gamma - size(\tau_1)_\Gamma)
\end{aligned}
$$

We are now ready to present the first of our main results.

Theorem 1 *During the execution of Curien and Ghelli's algorithm, if* $size(\sigma_1)_\Gamma \geq size(\tau_1)_\Gamma$ *(where* σ_1, τ_1, *and* Γ *are as defined in* NALL*) is true every time* NALL *is used to reduce a subtyping problem to the subtyping problem derived from the right template above the line* $(\Gamma, \alpha <: \tau_1 \vdash \sigma_2 <: \tau_2)$*, then the algorithm will terminate.*

Proof. Obvious from the results of this section. □

3.3 Subset restrictions to guarantee termination

When applying NALL on a subtyping problem, we shall require that we first consider the subtyping problem derived from $\Gamma \vdash \tau_1 <: \sigma_1$ (the left template above the line in NALL). Only if the algorithm terminates successfully on this problem do we consider the subtyping problem derived from $\Gamma, \alpha <: \tau_1 \vdash \sigma_2 <: \tau_2$ (the right template above the line in NALL). Hence, we can assume that $\Gamma \vdash \tau_1 <: \sigma_1$ when the algorithm is applied on the subtyping problem derived from $\Gamma, \alpha <: \tau_1 \vdash \sigma_2 <: \tau_2$. Assuming this, we have the following corollary to Theorem 1.

Corollary 1 *For every* τ, σ *that are bounds of polymorphic types and for every list of assumptions* Γ, *if* $\Gamma \vdash \tau <: \sigma \Rightarrow size(\tau)_\Gamma \leq size(\sigma)_\Gamma$, *then Curien and Ghelli's algorithm will terminate.*

Lemma 3 *If* $size(\tau_1)_\Gamma = size(\tau_2)_\Gamma$, *then, for any* σ, $size(\sigma)_{\Gamma, \alpha <: \tau_1} = size(\sigma)_{\Gamma, \alpha <: \tau_2}$.

Proof. This follows trivially from Lemma 2. $size(\sigma)_{\Gamma, \alpha <: \tau_1} = size(\sigma)_\Gamma + n(size(\tau_1)_\Gamma - 1) = size(\sigma)_\Gamma + n(size(\tau_2)_\Gamma - 1) = size(\sigma)_{\Gamma, \alpha <: \tau_2}$. □

We are now ready to present another key result, Theorem 2, that if the subset restrictions mentioned in Section 1 are satisfied, then the condition of Corollary 1 will be satisfied. A straightforward consequence of this is that if these subset restrictions are met, then Curien and Ghelli's algorithm will terminate, and hence completely bounded quantification is decidable.

Theorem 2 *For all types* τ, σ *that do not contain Top, and for any list of assumptions* Γ, *if* $\Gamma \vdash \tau <: \sigma$, *then* $size(\tau)_\Gamma = size(\sigma)_\Gamma$.

Proof. If $\Gamma \vdash \tau <: \sigma$, then there must be a proof $\Gamma^1 \vdash \tau^1 <: \sigma^1, \Gamma^2 \vdash \tau^2 <: \sigma^2, \ldots, \Gamma^n \vdash \tau^n <: \sigma^n$ where $\Gamma^n = \Gamma$, $\tau^n = \tau$, and $\sigma^n = \sigma$, and each $\Gamma^i \vdash \tau^i <: \sigma^i$ is either of the form of NREFL, or obtained from earlier steps in the proof using one of the rules NVAR, NARROW, or NALL. Note that NTOP will not be used in such a proof since *Top* does not occur in σ and τ.

We prove by induction that for all i $(1 \leq i \leq n)$, $size(\tau^i)_{\Gamma^i} = size(\sigma^i)_{\Gamma^i}$. The induction hypothesis is that for all j $(1 \leq j < i)$, $size(\tau^j)_{\Gamma^j} = size(\sigma^j)_{\Gamma^j}$. Assuming the induction hypothesis, we prove $size(\tau^i)_{\Gamma^i} = size(\sigma^i)_{\Gamma^i}$. There are four cases to consider:

1. $\Gamma^i \vdash \tau^i <: \sigma^i$ is of the form of NREFL. *i.e.*, $\tau^i = \sigma^i$. Therefore, $size(\tau^i)_{\Gamma^i} = size(\sigma^i)_{\Gamma^i}$.

2. $\Gamma^i \vdash \tau^i <: \sigma^i$ is derived using NVAR from an earlier step of the form $\Gamma^i \vdash \Gamma^i(\tau^i) <: \sigma^i$. In this case, τ^i is a variable bounded in Γ^i. Therefore, $size(\tau^i)_{\Gamma^i} = size(\Gamma^i(\tau^i))_{\Gamma^i} = size(\sigma^i)_{\Gamma^i}$.

3. $\Gamma^i \vdash \tau^i <: \sigma^i$ is derived using NARROW from earlier steps $\Gamma^k \vdash \tau^k <: \sigma^k$ and $\Gamma^l \vdash \tau^l <: \sigma^l$. Then $\Gamma^i = \Gamma^k = \Gamma^l$, $\tau^i = \sigma^k \rightarrow \tau^l$, and $\sigma^i = \tau^k \rightarrow \sigma^l$. Therefore, $size(\tau^i)_{\Gamma^i} = size(\sigma^k)_{\Gamma^i} + size(\tau^l)_{\Gamma^i} = size(\sigma^k)_{\Gamma^k} + size(\tau^l)_{\Gamma^l} = size(\tau^k)_{\Gamma^k} + size(\sigma^l)_{\Gamma^l} = size(\tau^k)_{\Gamma^i} + size(\sigma^l)_{\Gamma^i} = size(\sigma^i)_{\Gamma^i}$.

4. $\Gamma^i \vdash \tau^i <: \sigma^i$ is derived using NALL from earlier steps $\Gamma^k \vdash \tau^k <: \sigma^k$ and $\Gamma^l \vdash \tau^l <: \sigma^l$. Then $\Gamma^i = \Gamma^k$, $\Gamma^l = \Gamma^i, \alpha <: \tau^k$, $\tau^i = \forall \alpha <: \sigma^k.\tau^l$, and $\sigma^i = \forall \alpha <: \tau^k.\sigma^l$ for some variable α. Therefore, $size(\tau^i)_{\Gamma^i} = size(\sigma^k)_{\Gamma^i} + size(\tau^l)_{\Gamma^i, \alpha <: \sigma^k} = size(\sigma^k)_{\Gamma^k} + size(\tau^l)_{\Gamma^i, \alpha <: \tau^k} = size(\sigma^k)_{\Gamma^k} + size(\tau^l)_{\Gamma^l} = size(\tau^k)_{\Gamma^k} + size(\sigma^l)_{\Gamma^l} = size(\tau^k)_{\Gamma^i} + size(\sigma^l)_{\Gamma^i, \alpha <: \tau^k} = size(\sigma^i)_{\Gamma^i}$. □

4 Allowing unbounded quantification

We can relax our restriction on the use of *Top* to allow unbounded quantification and still retain decidability of subtype checking. Furthermore, any variable bounded by *Top* (directly or indirectly) may be used as a bound for another variable. We refer to these variables as "*Top*-bounded".

With this relaxation, there are two kinds of bounds that we can write: (1) Types that do not contain either *Top* or *Top*-bounded variables; and (2) Types that are either *Top* or a *Top*-bounded variable.

To show that this will not cause problems, we redefine *size*(*Top*) for all *Top*'s that occur as bounds of variables to be a number L that is larger than the *size* of any bounds of the first kind mentioned above. With this redefinition of *size*, it is quite easy to see that Corollary 1 continues to hold even when the bounds τ and σ are from this relaxed domain.

There are four cases to consider where τ and σ may each be either of the two kinds mentioned above. We consider each case separately:

1. τ and σ are both types not containing *Top* or *Top*-bounded variables: This case has been handled in Theorem 2.

2. τ and σ are both types that are either *Top* or a *Top*-bounded variable: Then $size(\tau) = size(\sigma) = L$.

3. τ is a type not containing *Top* or *Top*-bounded variables while σ is either *Top* or a *Top*-bounded variable: Then $size(\sigma) = L > size(\tau)$.

4. τ is either *Top* or a *Top*-bounded variable while σ is a type not containing *Top* or *Top*-bounded variables: In this case, τ cannot be a subtype of σ, so we need not consider it any further.

5 Record and union types

We have shown that one can achieve a decidable type system for a fairly unconstricted subset of $F_{\leq}$. An immediate extension that we started working on was the addition of record and union types. *However, it turns out that adding either record or union types to the type system (along with their associated inference rule) makes the subtyping problem undecidable. Finding the right restrictions to allow the addition of these types is the subject of current research.*

5.1 Record types

We denote the record type with fields $l_1 \ldots l_n$ having types $\tau_1 \ldots \tau_n$ respectively as $\{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}$. The inference rule for records is :

$$\text{(NREC)} \qquad \frac{\Gamma \vdash \sigma_1 <: \tau_1 \qquad \ldots \qquad \Gamma \vdash \sigma_n <: \tau_n}{\Gamma \vdash \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n, l_{n+1}{:}\sigma_{n+1}, \ldots\} <: \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}}$$

Essentially one can either add extra fields to a record type or specialize the types of existing fields to get a subtype.

As we saw earlier, the problem with the original unrestricted system was that one could have subtypes that were structurally much more complicated than the supertype and one could exploit this in creating subtyping subproblems that grew infinitely in their complexity. We prevented this finally by restricting the use of *Top*, which is what allowed a subtype to be more complicated than the supertype in the first place. However, record types provide another way of allowing a subtype to be more complex than the supertype, namely by adding extra fields. So the empty record type provides an entity conceptually similar to *Top*. We now show that one can reproduce the non-terminating example mentioned in Pierce [Pie92], in spite of the restrictions on *Top* with the use of records.

Let $\neg\tau = \tau \to b$ where b is some simple type. Note that $\neg\sigma <: \neg\tau$ iff $\tau <: \sigma$.

Let

$$\theta = \{a{:}\forall\alpha <: \{\}.\neg\{a{:}\forall\beta <: \alpha.\neg\beta\}\}$$

Now consider the subtyping problem

$$\alpha_0 <: \theta \vdash \alpha_0 <: \{a{:}\forall\alpha_1 <: \alpha_0.\neg\alpha_1\}$$

This causes the following sequence of subproblems to be generated infinitely:

$$\begin{array}{rcccl}
\alpha_0 <: \theta & \vdash & \alpha_0 & <: & \{a{:}\forall\alpha_1 <: \alpha_0.\neg\alpha_1\} \\
\alpha_0 <: \theta & \vdash & \{a{:}\forall\alpha_1 <: \{\}.\neg\{a{:}\forall\alpha_2 <: \alpha_1.\neg\alpha_2\}\} & <: & \{a{:}\forall\alpha_1 <: \alpha_0.\neg\alpha_1\} \\
\alpha_0 <: \theta, \alpha_1 <: \alpha_0 & \vdash & \neg\{a{:}\forall\alpha_2 <: \alpha_1.\neg\alpha_2\} & <: & \neg\alpha_1 \\
\alpha_0 <: \theta, \alpha_1 <: \alpha_0 & \vdash & \alpha_1 & <: & \{a{:}\forall\alpha_2 <: \alpha_1.\neg\alpha_2\} \\
\alpha_0 <: \theta, \alpha_1 <: \alpha_0 & \vdash & \alpha_0 & <: & \{a{:}\forall\alpha_2 <: \alpha_1.\neg\alpha_2\} \\
\alpha_0 <: \theta, \alpha_1 <: \alpha_0 & \vdash & \{a{:}\forall\alpha_2 <: \{\}.\neg\{a{:}\forall\alpha_3 <: \alpha_2.\neg\alpha_3\}\} & <: & \{a{:}\forall\alpha_2 <: \alpha_1.\neg\alpha_2\} \\
\alpha_0 <: \theta, \alpha_1 <: \alpha_0, \alpha_2 <: \alpha_1 & \vdash & \neg\{a{:}\forall\alpha_3 <: \alpha_2.\neg\alpha_3\} & <: & \neg\alpha_2
\end{array}$$

and so on.

This pattern of non-termination is practically identical to the one displayed in [Pie92].

5.2 Union types

Union types pose problems similar to that of record types.

Union types are written as $\{l_1:\tau_1 + \ldots + l_n:\tau_n\}$, with the usual meaning. The inference rule for union types is fairly straightforward.

$$(\text{NUNION}) \qquad \frac{\Gamma \vdash \sigma_1 <: \tau_1 \qquad \ldots \qquad \Gamma \vdash \sigma_n <: \tau_n}{\Gamma \vdash \{l_1:\sigma_1 + \ldots + l_n:\sigma_n\} <: \{l_1:\tau_1 + \ldots + l_n:\tau_n + \ldots\}}$$

The subtype is allowed to be a union over a subset of specializations of the types in the supertype.

Since the subtype can be a union of a fewer types than the supertype, one can have types with unions in contravariant positions thereby resulting in subtypes that have more complex structures than the supertype. Again, one can exploit this to generate an example of a subtyping problem that doesn't terminate. Behavior identical to that displayed in the previous example on record types arises for the subtyping problem

$$\alpha_0 <: \theta \vdash \alpha_0 <: \neg\{a:\neg(\forall\alpha_1 <: \alpha_0.\neg\alpha_1)\}$$

where

$$\theta = \neg\{a:\neg(\forall\alpha <: \neg\{\}.\{a:\neg(\forall\beta <: \alpha.\neg\beta)\})\}$$

6 Future work

We are currently working on various possible subset restrictions on records and unions to make subtype checking decidable in the presence of these types. Following this, we intend to extend our system with recursive types. This might involve techniques similar to those employed by Amadio and Cardelli [AC91] to study the interaction of recursive types with subtyping. A further extension would be the addition of the so-called F-bounds [CCH*89], which essentially allow the bounds of polymorphic functions to be recursive.

Acknowledgements

We would like to thank John Mitchell for many insightful discussions. We are also grateful to David Luckham, Neel Madhav, Sigurd Meldal and other members of the Programming and Verification Group at Stanford for many useful discussions on the type system for *Rapide*. The authors were supported by DARPA grant ONR N00014-90-J1232 (Sriram) and NSF grant CCR-8814921 (Dinesh).

References

[AC91] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 104–118, 1991.

[BL90] F. Belz and D. C. Luckham. A new approach to prototyping Ada-based hardware/software systems. In *Proc. ACM Tri-Ada Conference*, pages 141–155, 1990.

[CCH*89] P. Canning, W. Cook, W. Hill, J. C. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[CG] P.-L. Curien and G. Ghelli. Coherence of subsumption. (to appear).

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Ghe90] G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, University of Pisa, 1990.

[MMM91] J. C. Mitchell, S. Meldal, and N. Madhav. An extension of standard ML modules with subtyping and inheritance. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 270–278, 1991.

[Pie92] B. Pierce. Bounded quantification is undecidable. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, pages 305–315, 1992.

An Extension of ML with First-Class Abstract Types

Konstantin Läufer,* New York University, laufer@cs.nyu.edu
Martin Odersky,† Yale University, odersky@cs.yale.edu

1 Introduction

Many statically-typed programming languages provide an abstract data type construct, such as the package in Ada, the cluster in CLU, and the module in Modula2. In these languages, an abstract data type consists of two parts, interface and implementation. The implementation consists of one or more representation types and some operations on these types; the interface specifies the names and types of the operations accessible to the user of the abstract data type.

ML [MTH90] provides two distinct constructs for describing abstract data types:

- The (obsolete) **`abstype`** mechanism is used to declare an abstract data type with a single implementation. It has been superseded by the module system.
- The ML module system provides signatures, structures, and functors. Signatures act as interfaces of abstract data types and structures as their implementations; functors are essentially parametrized structures. Several structures may share the same signature, and a single structure may satisfy several signatures. However, structures are not first-class values in ML for type-theoretic reasons discussed in [Mac86] [MH88]. This leads to considerable difficulties in a number of practical programming situations.

Mitchell and Plotkin show that abstract types can be given existential type [MP88]. By stating that a value v has the existential type $\exists\alpha.\tau$, we mean that for some fixed, unknown type $\hat{\tau}$, v has type $\tau\,[\hat{\tau}/\alpha]$. This paper presents a semantic extension of ML, where the component types of a datatype may be existentially quantified. We show how datatypes over existential types add significant flexibility to the language without even changing ML syntax; in particular, we give examples demonstrating how we express

- first-class abstract types,
- multiple implementations of a given abstract type,
- heterogeneous aggregates of different implementations of the same abstract type, and
- dynamic dispatching of operations with respect to the implementation type.

We have a deterministic Damas-Milner inference system [DM82] [CDDK86] for our language, which leads to a syntactically sound and complete type reconstruction algorithm. Furthermore, the type system is semantically sound with respect to a standard denotational semantics.

Most previous work on existential types does not consider type reconstruction. Other work appears to be semantically unsound or does not permit polymorphic instantiation of variables of existential type. By contrast, in our system such variables are **`let`**-bound and may be instantiated polymorphically.

We have implemented a Standard ML prototype of an interpreter with type reconstruction for our core language, Mini-ML [CDDK86] extended with recursive datatypes over existentially quantified component types. All examples from this paper have been developed and tested using our interpreter.

* Supported by the Defence Advanced Research Project Agency/Information Systems Technology Office under the Office of Naval Research contract N00014-91-5-1472
† Supported by the Defence Advanced Research Project Agency/Information Systems Technology Office under the Office of Naval Research contract N00014-91-J-4043

2 ML Datatypes with Existential Component Types

In ML, datatype declarations are of the form

datatype [*arg*] T = K_1 **of** τ_1 | ... | K_n **of** τ_n

where the K's are value constructors and the optional prefix argument *arg* is used for formal type parameters, which may appear free in the component types τ_i. The value constructor functions are universally quantified over these type parameters, and no other type variables may appear free in the τ_i's.

An example for an ML datatype declaration is

```
datatype 'a Mytype = mycons of 'a * ('a -> int)
```

Without altering the syntax of the datatype declaration, we now give a meaning to type variables that appear free in the component types, but not in the type parameter list. We interpret such type variables as existentially quantified.

For example,

```
datatype Key = key of 'a * ('a -> int)
```

describes a datatype with one value constructor whose arguments are pairs of a value of type **'a** and a function from type **'a** to **int**. The question is what we can say about **'a**. The answer is, nothing, except that the value is of the same type **'a** as the function domain. To illustrate this further, the type of the expression

```
key(3,fn x => 5)
```

is **Key**, as is the type of the expression

```
key([1,2,3],length)
```

where **length** is the built-in function on lists. Note that no argument types appear in the result type of the expression. On the other hand,

```
key(3,length)
```

is not type-correct, since the type of 3 is different from the domain type of **length**.

We recognize that **Key** is an abstract type comprised by a value of some type and an operation on that type yielding an **int**. It is important to note that values of type **Key** are first-class; they may be created dynamically and passed around freely as function parameters. The two different values of type **Key** in the previous examples may be viewed as two different implementations of the same abstract type.

Besides constructing values of datatypes with existential component types, we can decompose them using the **let** construct. We impose the restriction that no type variable that is existentially quantified in a **let** expression appears in the result type of this expression or in the type of a global identifier. Analogous restrictions hold for the corresponding **open** and **abstype** constructs described in [CW85] [MP88].

For example, assuming **x** is of type **Key**, then

```
let val key(v,f) = x in
    f v
end
```

has a well-defined meaning, namely the **int** result of **f** applied to **v**. We know that this application is type-safe because the pattern matching succeeds, since **x** was constructed using constructor **key**, and at that time it was enforced that **f** can safely be applied to **v**. On the other hand,

```
let val key(v,f) = x in
    v
end
```

is not type-correct, since we do not know the type of **v** statically and, consequently, cannot assign a type to the whole expression.

Our extension to ML allows us to deal with existential types as described in [CW85] [MP88], with the further improvement that decomposed values of existential type are **let**-bound and may be instantiated polymorphically. This is illustrated by the following example,

```
datatype 'a t = k of ('a -> 'b) * ('b -> int)
let val k(f1,f2) = k(fn x => x,fn x => 3) in
    (f2(f1 7),f2(f1 true))
end
```

which results in `(3,3)`. In most previous work, the value on the right-hand side of the binding would have to be bound and decomposed twice.

3 Some Motivating Examples

Minimum over a heterogeneous list

Extending on the previous example, we first show how we construct heterogeneous lists over different implementations of the same abstract type and define functions that operate uniformly on such heterogeneous lists. A heterogeneous list of values of type **Key** could be defined as follows:

```
val hetlist =
    [key(3,fn x => x), key([1,2,3,4],length), key(7,fn x => 0),
     key(true,fn x => if x then 1 else 0), key(12,fn x => 3)]
```

The type of **hetlist** is **Key list**; it is a homogeneous list of elements each of which could be a different implementation of type **Key**. We define the function **min**, which finds the minimum of a list of **Key**'s with respect to the integer value obtained by applying the second component (the function) to the first component (the value).

```
fun min [x] = x
  | min ((key(v1,f1))::xs) =
        let val key(v2,f2) = min xs in
            if f1 v1 <= f2 v2 then key(v1,f1) else key(v2,f2)
        end
```

Then **min hetlist** returns **key(7,fn x => 0)**, the third element of the list.

Stacks parametrized by element type

The previous examples involved datatypes with existential types but without polymorphic type parameters. As an example for a type involving both, we show an abstract stack parametrized by element type.

```
datatype 'a Stack = stack of {value   : 'b,
                              empty   : 'b,
                              push    : 'a * 'b -> 'b
                              pop     : 'b -> 'a * 'b
                              top     : 'b -> 'a,
                              isempty : 'b -> bool}
```

An implementation of an **int Stack** in terms of the built-in type **list** can be given as

```
stack{value = [1,2,3], empty = [], push = op ::,
      pop = fn xs => (hd xs,tl xs), top = hd, isempty = null}
```

An alternative implementation of **Stack** could be given, among others, based on arrays. Different implementations could then be combined in a list of stacks. To facilitate dynamic dispatching, constructors of stacks of different implementations can be provided together with stack operations that work uniformly

across implementations. These "outer" operations work by opening the stack, applying the intended "inner" operation, and encapsulating the stack again, for example

```
fun makeliststack xs = stack{value = xs, empty = [],push = op ::,
        pop = fn xs => (hd xs,tl xs), top = hd, isempty = null}
fun makearraystack xs = stack{...}
fun push a (stack{value = v, push = pu, empty = e,
                  pop = po, top = t, isempty = i}) =
           stack{value = pu(a,v), push = pu, empty = e,
                  pop = po, top = t, isempty = i}
map (push 8) [makeliststack [2,4,6], makearraystack [3,5,7]]
```

4 Type-Theoretical Aspects

A deterministic type inference system for our language is given in the appendix; it leads directly to a syntactically sound and complete type reconstruction algorithm to compute principal types. Our type system is semantically sound with respect to a standard denotational semantics. Moreover, it is a conservative extension of ML. That is, for a program in our language whose declarations introduce no existentially quantified type variables, our type reconstruction algorithm and the ML type reconstruction algorithm compute the same type. A comprehensive treatment of polymorphic type inference with existential types is found in [Lä92].

5 Related Work

Hope+C

The only other work known to us that deals with Damas-Milner-style type inference for existential types is [Per90]. However, the typing rules given there are not sufficient to guarantee the absence of runtime type errors, even though the Hope+C compiler seems to impose sufficient restrictions. The following unsafe program, here given in ML syntax, is well-typed according to the typing rules, but rejected by the compiler:

```
datatype T = K of ''a
fun f x = let val K z = x in z end
f(K 1) = f(K true)
```

XML$^+$

The possibility of making ML structures first-class by implicitly hiding their type components is discussed in [MMM91] without addressing the issue of type inference. By hiding the type components of a structure, its type is implicitly coerced from a strong sum type to an existential type. Detailed discussions of sum types can be found in [Mac86] [MH88].

Haskell with existential types

Existential types combine well with the systematic overloading polymorphism provided by Haskell type classes [WB89]; this point is further discussed in [LO91]. Briefly, we extend Haskell's data declaration in a similar way as the ML datatype declaration above. In Haskell [HPW91], it is possible to specify what type class a (universally quantified) type variable belongs to. In our extension, we can do the same for existentially quantified type variables. This lets us construct heterogeneous aggregates over a given type class.

Dot notation

MacQueen [Mac86] observes that the use of existential types in connection with an elimination construct (**open**, **abstype**, or our **let**) is impractical in certain programming situations; often, the scope of the elimination construct has to be made so large that some of the benefits of abstraction are lost. A formal treatment of the dot notation, an alternative used in actual programming languages, is found in [CL90]. An extension of ML with an analogous notation is described in [Lä92].

Dynamics in ML

An extension of ML with objects that carry dynamic type information is described in [LM91]. A dynamic is a pair consisting of a value and the type of the value. Such an object is constructed from a value by applying the constructor **dynamic**. The object can then be dynamically coerced by pattern matching on both the value and the runtime type. Existential types are used to match dynamic values against dynamic patterns with incomplete type information. Dynamics are useful for typing functions such as **eval**. However, they do not provide type abstraction, since they give access to the type of an object at runtime. It seems possible to combine their system with ours, extending their existential patterns to existential types. We are currently investigating this point.

Acknowledgments

We would like to express our thanks to Ben Goldberg, Fritz Henglein, Ross Paterson, Nigel Perry, Benjamin Pierce, and Phil Wadler, for helpful suggestions and stimulating discussions.

A Formal Discussion of the Extended Language

In this appendix, we describe the formal language and the type system underlying our extension of ML. The typing rules and auxiliary functions translate to the type reconstruction algorithm given below.

A.1 Syntax

Language syntax

Identifiers	x	
Constructors	K	
Expressions	$e ::=$	$x \mid (e_1, e_2) \mid e\ e' \mid \lambda x.e \mid$ **let** $x = e$ **in** $e' \mid$
		data $\forall\alpha_1 \ldots \alpha_n.\chi$ **in** $e \mid K \mid$ **is** $K \mid$ **let** $K\ x = e$ **in** e'

In addition to the usual constructs (identifiers, applications, λ-abstractions, and **let** expressions), we introduce desugared versions of the ML constructs that deal with datatypes. A **data** declaration defines a new datatype; values of this type are created by applying a constructor K, their tags can be inspected using an **is** expression, and they can be decomposed by a pattern-matching **let** expression. The following example shows a desugared definition of ML's list type and the associated length function.

datatypes. A **data** declaration defines a new datatype; values of this type are created by applying a constructor K, their tags can be inspected using an **is** expression, and they can be decomposed by a pattern-matching **let** expression. The following example shows a desugared definition of ML's list type and the associated length function.

```
data ∀α.(μβ.Nil unit + Cons α×β) in
    let length = fix λlength.λxs.
                     if (is Nil xs)
                        0
                        (let Cons ab = xs in + (length(snd ab)) 1)
    in
        length(Cons(3,Cons(7,Nil())))
```

Type syntax

Type variables	α	
Skolem functions	κ	

Types	τ ::=	$unit \mid bool \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau \to \tau' \mid \kappa(\tau_1, \ldots, \tau_n) \mid \chi$
Recursive types	χ ::=	$\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$ where $K_i \neq K_j$ if $i \neq j$
Existential types	η ::=	$\exists\alpha.\eta \mid \tau$
Type schemes	σ ::=	$\forall\alpha.\sigma \mid \tau$
Assumptions	a ::=	$\sigma/x \mid \forall\alpha_1 \ldots \alpha_n.\chi/K$

Our type syntax includes recursive types χ and Skolem type constructors κ; the latter are used to type identifiers bound by a pattern-matching **let** whose type is existentially quantified. Explicit existential types arise only as domain types of value constructors. Assumption sets serve two purposes: they map identifiers to type schemes and constructors to the recursive type schemes they belong to. Thus, when we write $A(K)$, we mean the σ such that $\sigma = \forall\alpha_1 \ldots \alpha_n.\ldots + K\eta + \ldots$. Further, let $\Sigma[K\eta]$ stand for sum type contexts such as $K_1\eta_1 + \ldots + K_m\eta_m$, where $K_i = K$ and $\eta_i = \eta$ for some i.

A.2 Type Inference

Instantiation and generalization of type schemes

$\forall\alpha_1 \ldots \alpha_n.\tau \geq \tau'$	iff there are types $\tau_1, \ldots \tau_n$ such that $\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$
$\exists\alpha_1 \ldots \alpha_n.\tau \leq \tau'$	iff there are types $\tau_1, \ldots \tau_n$ such that $\tau' = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$
$gen(A, \tau)$	$= \forall(FV(\tau) \setminus FV(A)).\tau$
$skolem(A, \exists\gamma_1 \ldots \gamma_n.\tau)$	$= \tau[\kappa_i(\alpha_1, \ldots \alpha_k)/\gamma_i]$ where $\kappa_1 \ldots \kappa_n$ are new Skolem type constructors such that $\{\kappa_1, \ldots, \kappa_n\} \cap FS(A) = \emptyset$, and $\{\alpha_1, \ldots, \alpha_k\} = FV(\exists\gamma_1 \ldots \gamma_n.\tau) \setminus FV(A)$

The first three auxiliary functions are standard. The function *skolem* replaces each existentially quantified variable in a type by a unique type constructor whose actual arguments are those free variables of the type that are not free in the assumption set; this reflects the "maximal" knowledge we have about the type represented by an existentially quantified type variable. In addition to FV, the set of free type variables in a type scheme or assumption set, we use FS, the set of Skolem type constructors that occur in a type scheme or assumption set.

Inference rules for expressions

The first five typing rules are essentially the same as in [CDDK86].

(VAR) $$\frac{A(x) \geq \tau}{A \vdash x : \tau}$$

(PAIR) $$\frac{A \vdash e_1 : \tau_1 \qquad A \vdash e_2 : \tau_2}{A \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\text{(APPL)}\quad \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash e\ e' : \tau}$$

$$\text{(ABS)}\quad \frac{A[\tau'/x] \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$$

$$\text{(LET)}\quad \frac{A \vdash e : \tau \qquad A[gen(A,\tau)/x] \vdash e' : \tau'}{A \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \tau'}$$

The new rules DATA, CONS, TEST, and PAT are used to type datatype declarations, value constructors, **is** expressions, and pattern-matching **let** expressions, respectively.

$$\text{(DATA)}\quad \frac{\sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m \qquad FV(\sigma) = \varnothing \qquad A[\sigma/K_1, \ldots, \sigma/K_m] \vdash e : \tau}{A \vdash \texttt{data}\ \sigma\ \texttt{in}\ e : \tau}$$

The DATA rule elaborates a declaration of a recursive datatype. It checks that the type scheme is closed and types the expression under the assumption set extended with assumptions about the constructors.

$$\text{(CONS)}\quad \frac{A(K) \geq \mu\beta.\Sigma[K\eta] \qquad \eta[\mu\beta.\Sigma[K\eta]/\beta] \leq \tau}{A \vdash K : \tau \to \mu\beta.\Sigma[K\eta]}$$

The CONS rule observes the fact that existential quantification in argument position means universal quantification over the whole function type; this is expressed by the second premise.

$$\text{(TEST)}\quad \frac{A(K) \geq \mu\beta.\Sigma[K\eta]}{A \vdash \texttt{is}\ K : (\mu\beta.\Sigma[K\eta]) \to bool}$$

The TEST rule ensures that **is** K is applied only to arguments whose type is the same as the result type of constructor K.

$$\text{(PAT)}\quad \frac{A \vdash e : \mu\beta.\Sigma[K\eta] \qquad FS(\tau') \subseteq FS(A) \qquad A[gen(A, skolem(A, \eta[\mu\beta.\Sigma[K\eta]/\beta]))/x] \vdash e' : \tau'}{A \vdash \texttt{let}\ K\ x = e\ \texttt{in}\ e' : \tau'}$$

The last rule, PAT, governs the typing of pattern-matching **let** expressions. It requires that the expression e be of the same type as the result type of the constructor K. The body e' is typed under the assumption set extended with an assumption about the bound identifier x. By definition of the function *skolem*, the new Skolem type constructors do not appear in A; this ensures that they do not appear in the type of any identifier free in e' other than x. It is also guaranteed that the Skolem constructors do not appear in the result type τ'.

Relation to the ML Type Inference System

Theorem 1 [Conservative extension] Let Mini-ML' be an extension of Mini-ML with recursive datatypes, but not with existential quantification. Then, for any Mini-ML' expression e, $A \vdash e : \tau$ iff $A \vdash_{\text{Mini-ML'}} e : \tau$.

Proof: By structural induction on e.

Corollary 2 [Conservative extension] Our type system is a conservative extension of the Mini-ML type sys-

tem described in [CDDK86], in the following sense: For any Mini-ML expression e, $A \vdash e : \tau$ iff $A \vdash_{\text{Mini-ML}} e : \tau$.

Proof: Follows immediately from the previous theorem.

A.3 Type Reconstruction

The type reconstruction algorithm is a straightforward translation from the deterministic typing rules, using a standard unification algorithm [Rob65] [MM82]. We conjecture that its complexity is the same as that of algorithm W.

Auxiliary functions

In our algorithm, we need to instantiate universally quantified types and generalize existentially quantified types. Both are handled in the same way.

$inst_\forall (\forall\alpha_1 \ldots \alpha_n.\tau)$ $= \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]$ where $\beta_1, \ldots, \beta_n$ are fresh type variables

$inst_\exists (\exists\alpha_1 \ldots \alpha_n.\tau)$ $= \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]$ where $\beta_1, \ldots, \beta_n$ are fresh type variables

The functions *skolem* and *gen* are the same as in the inference rules, with the additional detail that *skolem* always creates fresh Skolem type constructors.

Algorithm

Our type reconstruction function takes an assumption set and an expression, and it returns a substitution and a type expression. There is one case for each typing rule.

$TC(A, x) = (Id, inst_\forall (A(x)))$

$TC(A, (e_1, e_2)) =$ **let** $(S_1, \tau_1) = TC(A, e_1)$
$(S_2, \tau_2) = TC(S_1 A, e_2)$
in $(S_2 S_1, S_2 \tau_1 \times \tau_2)$

$TC(A, ee') =$ **let** $(S, \tau) = TC(A, e)$
$(S', \tau') = TC(SA, e')$
β be a fresh type variable
$U = mgu(S'\tau, \tau' \rightarrow \beta)$
in $(US'S, U\beta)$

$TC(A, \lambda x.e) =$ **let** β be a fresh type variable
$(S, \tau) = TC(A[\beta/x], e)$
in $(S, S\beta \rightarrow \tau)$

$TC(A,$ **let** $x = e$ **in** $e') =$ **let** $(S, \tau) = TC(A, e)$
$(S', \tau') = TC(SA[gen(SA, \tau)/x], e')$
in $(S'S, \tau')$

$$
\begin{aligned}
TC(A, \textbf{data}\ \sigma\ \textbf{in}\ e) \quad &= \textbf{let}\ \ \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m = \sigma\ \textbf{in} \\
&\qquad \textbf{if}\ \ FV(\sigma) = \emptyset\ \textbf{then} \\
&\qquad\quad TC(A[\sigma/K_1, \ldots, \sigma/K_m], e)
\end{aligned}
$$

$$
\begin{aligned}
TC(A, K) \quad &= \textbf{let}\ \ \tau = inst_\forall(A(K)) \\
&\qquad\quad \mu\beta.\ldots + K\eta + \ldots = \tau \\
&\quad \textbf{in}\ \ (Id, (inst_\exists(\eta)) \rightarrow \tau)
\end{aligned}
$$

$$
\begin{aligned}
TC(A, \textbf{is}\ K) \quad &= \textbf{let}\ \ \tau = inst_\forall(A(K)) \\
&\quad \textbf{in}\ \ (Id, \tau \rightarrow bool)
\end{aligned}
$$

$$
\begin{aligned}
TC(A, \textbf{let}\ K\ x = e\ \textbf{in}\ e') \quad &= \textbf{let}\ \ \hat{\tau} = inst_\forall(A(K)) \\
&\qquad\quad \mu\beta.\ldots + K\eta + \ldots = \hat{\tau} \\
&\qquad\quad (S, \tau) = TC(A, e) \\
&\qquad\quad U = (mgu(\hat{\tau}, \tau))S \\
&\qquad\quad \tau_K = skolem(UA, U\eta) \\
&\qquad\quad (S', \tau') = TC(UA[gen(UA, \tau_K)/x], e') \\
&\quad \textbf{in} \\
&\qquad \textbf{if}\ \ FS(\tau') \subseteq FS(S'UA) \wedge \\
&\qquad\quad (FS(\tau_K) \setminus FS(U\eta)) \cap FS(S'UA) \neq \emptyset \\
&\qquad \textbf{then}\ \ (S'U, \tau')
\end{aligned}
$$

Theorem 3 [Syntactic Soundness and Completeness] The type reconstruction algorithm TC is sound and complete with respect to the type inference relation $\vdash$.

Proof: We extend the proof given in [CDDK86] to deal with the new constructs.

A.4 Semantics

We give a standard denotational semantics. The evaluation function E maps an expression $e \in Exp$ to some semantic value v, in the context of an evaluation environment $\rho \in Env$. An evaluation environment is a partial mapping from identifiers to semantic values. Runtime type errors are represented by the special value wrong. Tagged values are used to capture the semantics of algebraic data types.

We distinguish between the three error situations, runtime type errors (wrong), nontermination, and a mismatch when an attempt is made to decompose a tagged value whose tag does not match the tag of the destructor. Both nontermination and mismatch are expressed by $\perp$.

Our type inference system is sound with respect to the evaluation function; a well-typed program never evaluates to wrong. The formal proof for semantic soundness is given below.

It should be noted that we do not commit ourselves to a strict or non-strict evaluation function. Therefore, our treatment of existential types applies to languages with both strict and non-strict semantics. For either case, appropriate conditions would have to be added to the definition of the evaluation function.

Semantic domain

Unit value	$U = \{\mathsf{unit}\}_\perp$
Boolean values	$B = \{\mathsf{false}, \mathsf{true}\}_\perp$
Constructor tags	C
Semantic domain	$V \cong U + B + (V \to V) + (V \times V) + (C \times V) + \{\mathsf{wrong}\}_\perp$

In the latter definition of V, $+$ stands for the coalesced sum, so that all types over V share the same $\perp$.

Semantics of expressions

The semantic function for expressions,

$$E : Exp \to Env \to V,$$

is defined as follows:

$$\begin{aligned}
E[\![x]\!]\rho &= \rho(x) \\
E[\![(e_1, e_2)]\!]\rho &= \langle E[\![e_1]\!]\rho, E[\![e_2]\!]\rho \rangle \\
E[\![e\,e']\!]\rho &= \mathsf{if}\ E[\![e]\!]\rho \in V \to V\ \mathsf{then} \\
&\quad (E[\![e]\!]\rho)(E[\![e']\!]\rho) \\
&\quad \mathsf{else\ wrong} \\
E[\![\lambda x.e]\!]\rho &= \lambda v \in V.E[\![e]\!](\rho[v/x]) \\
E[\![\mathbf{let}\ x = e\ \mathbf{in}\ e']\!]\rho &= E[\![e']\!](\rho[E[\![e]\!]\rho/x]) \\
E[\![\mathbf{data}\ \sigma\ \mathbf{in}\ e]\!]\rho &= E[\![e]\!]\rho \\
E[\![K]\!]\rho &= \lambda v \in V.\langle K, v \rangle \\
E[\![\mathbf{is}\ K]\!]\rho &= \lambda v \in V.\mathsf{if}\ v \in \{K\} \times V\ \mathsf{then\ true\ else\ false} \\
E[\![\mathbf{let}\ K\,x = e\ \mathbf{in}\ e']\!]\rho &= E[\![e']\!](\rho[\mathsf{if}\ E[\![e]\!]\rho \in \{K\} \times V\ \mathsf{then} \\
&\qquad \mathsf{snd}(E[\![e]\!]\rho) \\
&\qquad \mathsf{else}\ \perp/x])
\end{aligned}$$

Semantics of types

Following [MPS86], we identify types with *weak ideals* over the semantic domain V. A type environment $\psi \in TEnv$ is a partial mapping from type variables to ideals and from Skolem type constructors to functions between ideals. The semantic interpretation of types,

$$T : TExp \to TEnv \to \Im(V)$$

is defined as follows.

$$
\begin{aligned}
T[\![\mathit{unit}]\!]\psi &= U \\
T[\![\mathit{bool}]\!]\psi &= B \\
T[\![\alpha]\!]\psi &= \psi(\alpha) \\
T[\![\tau_1 \times \tau_2]\!]\psi &= T[\![\tau_1]\!]\psi \times T[\![\tau_2]\!]\psi \\
T[\![\tau \to \tau']\!]\psi &= T[\![\tau]\!]\psi \to T[\![\tau']\!]\psi \\
T[\![\kappa(\tau_1, \ldots, \tau_n)]\!]\psi &= (\psi(\kappa))\,(T[\![\tau_1]\!]\psi, \ldots, T[\![\tau_n]\!]\psi) \\
T[\![\mu\beta.\textstyle\sum K_i\eta_i]\!]\psi &= \mu(\lambda I \in \mathfrak{I}(V).\textstyle\sum\{K_i\} \times T[\![\eta_i]\!](\psi[I/\beta])) \\
T[\![\forall\alpha.\sigma]\!]\psi &= \bigcap_{I \in \mathfrak{R}} \lambda I \in \mathfrak{I}(V).T[\![\sigma]\!](\psi[I/\alpha]) \\
T[\![\exists\alpha.\eta]\!]\psi &= \bigsqcup_{I \in \mathfrak{R}} \lambda I \in \mathfrak{I}(V).T[\![\eta]\!](\psi[I/\alpha])
\end{aligned}
$$

The universal and existential quantifications range over the set $\mathfrak{R} \subseteq \mathfrak{I}(V)$ of all ideals that do not contain wrong. Note that the sum in the definition of recursive types is actually a union, since the constructor tags are assumed to be distinct. It should also be noted that our interpretation does not handle ML's nonregular, mutually recursive datatypes; it appears that the PER model described in [BM92] would provide an adequate interpretation.

Theorem 4 The semantic function for types is well-defined.

Proof: As in [MPS86]. We observe that $\lambda I \in \mathfrak{I}(V).\sum\{K_i\} \times T[\![\eta_i]\!](\psi[I/\alpha])$ is always contractive, since cartesian product and sum of ideals are contractive; therefore, the fixed point of such a function exists.

Lemma 5 Let ψ be a type environment such that for every $\alpha \in \mathrm{Dom}\,\psi$, $\mathsf{wrong} \notin \psi(a)$. Then for every type scheme σ, $\mathsf{wrong} \notin T[\![\sigma]\!]\psi$.

Proof: By structural induction on σ.

Lemma 6 [Substitution] $T[\![\sigma\,[\sigma'/\alpha]]\!]\psi = T[\![\sigma]\!](\psi[T[\![\sigma']\!]\psi/\alpha])$.

Proof: Again, by structural induction on σ.

Definition 1 [Semantic type judgment] Let A be an assumption set, e an expression, and σ a type scheme. We define $\models_{\rho,\psi} A$ as meaning that $\mathrm{Dom}\,A \subseteq \mathrm{Dom}\,\rho$ and for every $x \in \mathrm{Dom}\,A$, $\rho(x) \in T[\![A(x)]\!]\psi$; further, we say $A \models_{\rho,\psi} e : \sigma$ iff $\models_{\rho,\psi} A$ implies $E[\![e]\!]\rho \in T[\![\sigma]\!]\psi$; and finally, $A \models e : \sigma$ means that for all $\rho \in \mathit{Env}$ and $\psi \in \mathit{TEnv}$ we have $A \models_{\rho,\psi} e : \sigma$.

Theorem 7 [Semantic Soundness] If $A \vdash e : \tau$ then $A \models e : \tau$.

Proof: By induction on the size of the proof tree for $A \vdash e : \tau$. We need to consider each of the cases given by the type inference rules. Applying the inductive assumption and the typing judgments from the preceding

steps in the type derivation, we use the semantics of the types of the partial results of the evaluation. In each of the cases below, choose ψ and ρ arbitrarily, such that $\models_{\rho,\psi} A$. We include only the nonstandard cases. Lemma 6 will be used with frequency.

$A \vdash$ **data** $\forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$ **in** $e : \tau$

The premise in the type derivation is $A\,[\sigma/K_1, \ldots, \sigma/K_m] \vdash e : \tau$, where $\sigma = \forall\alpha_1 \ldots \alpha_n.\mu\beta.K_1\eta_1 + \ldots + K_m\eta_m$. Since by definition, $\models_{\rho,\psi} A\,[\sigma/K_1, \ldots, \sigma/K_m]$, we can use the inductive assumption to obtain $E[\![$ **data** $\forall\alpha_1 \ldots \alpha_n.\chi$ **in** $e]\!]\,\rho = E[\![e]\!]\,\rho \in T[\![\tau]\!]\,\psi$.

$A \vdash K : \tau \rightarrow \mu\beta.\Sigma[K\eta]$

The last premise in the type derivation is $\eta\,[\mu\beta.\Sigma[K\eta]/\beta] \leq \tau$, where $\eta = \exists\gamma_1 \ldots \gamma_n.\hat{\tau}$. By definition of instantiation of existential types, $\tau = \hat{\tau}\,[\tau_j/\gamma_j, \mu\beta.\Sigma[K\eta]/\beta]$ for some types $\tau_1, \ldots, \tau_n$. First, choose an arbitrary $v \in T[\![\tau]\!]\,\psi$ and a finite $a \leq v$. Now,

$$
\begin{aligned}
a &\in (T[\![\hat{\tau}\,[\tau_j/\gamma_j, \mu\beta.\Sigma[K\eta]/\beta]]\!]\,\psi)^\circ \\
&= (T[\![\hat{\tau}\,[\mu\beta.\Sigma[K\eta]/\beta]]\!]\,(\psi\,[T[\![\tau_j]\!]\,\psi/\gamma_j]))^\circ \\
&\subseteq \bigcup_{J_1, \ldots, J_n \in \mathfrak{R}} (T[\![\hat{\tau}\,[\mu\beta.\Sigma[K\eta]/\beta]]\!]\,(\psi\,[J_j/\gamma_j]))^\circ \\
&= \Big(\bigsqcup_{J_1, \ldots, J_n \in \mathfrak{R}} T[\![\hat{\tau}\,[\mu\beta.\Sigma[K\eta]/\beta]]\!]\,(\psi\,[J_j/\gamma_j])\Big)^\circ \\
&= (T[\![\eta\,[\mu\beta.\Sigma[K\eta]/\beta]]\!]\,\psi)^\circ.
\end{aligned}
$$

Hence, $v = \bigsqcup \{a \mid a \text{ finite and } a \leq v\} \in T[\![\eta\,[\mu\beta.\Sigma[K\eta]/\beta]]\!]\,\psi$, by closure of ideals under limits. Consequently,

$$
\begin{aligned}
\langle K, v\rangle &\in \{K\} \times T[\![\eta\,[\mu\beta.\Sigma[K\eta]/\beta]]\!]\,\psi \\
&\subseteq \ldots + \{K\} \times T[\![\eta\,[\mu\beta.\Sigma[K\eta]/\beta]]\!]\,\psi + \ldots \\
&= \ldots + \{K\} \times T[\![\eta]\!]\,(\psi[T[\![\mu\beta.\Sigma[K\eta]]\!]\,\psi/\beta]) + \ldots \\
&= T[\![\mu\beta.\Sigma[K\eta]]\!]\,\psi.
\end{aligned}
$$

Hence $E[\![K]\!]\,\rho \in T[\![\tau \rightarrow \mu\beta.\Sigma[K\eta]]\!]\,\psi$.

$A \vdash$ **is** $K : (\mu\beta.\Sigma[K\eta]) \rightarrow bool$

Choose an arbitrary $v \in T[\![\mu\beta.\Sigma[K\eta]]\!]\,\psi$. Clearly, $(E[\![$**is** $K]\!]\,\rho)\,v \in B$, whence $E[\![$**is** $K]\!]\,\rho \in T[\![(\mu\beta.\Sigma[K\eta]) \rightarrow bool]\!]\,\psi$.

$A \vdash$ **let** $K\,x = e$ **in** $e' : \tau'$

We follow the proof in [MPS86]. The first premise in the type derivation is $A \vdash e : \tau$, where $\tau = \mu\beta.\Sigma[K\eta]$ and $\eta = \exists\gamma_1 \ldots \gamma_n.\hat{\tau}$. Let $\{\alpha_1, \ldots, \alpha_k\} = FV(\tau) \setminus FV(A)$. Then, for every $I_1, \ldots, I_k \in \mathfrak{I}(V)$, $\models_{\rho,\psi[I_i/\alpha_i]} A$ holds, since none of the α_i's are free in A.

Let $v = E[\![e]\!]\,\rho$; by the inductive assumption, $v \in T[\![\tau]\!]\,(\psi\,[I_i/\alpha_i])$. Consequently,

$$
v \in \bigcap_{I_1, \ldots, I_k \in \mathfrak{R}} T[\![\tau]\!]\,(\psi\,[I_i/\alpha_i])
$$

$$= \bigcap_{I_1, \ldots, I_k \in \mathfrak{R}} T[\![\mu\beta.\Sigma[K\eta]]\!] \ (\psi\,[I_i/\alpha_i])$$

$$= \ldots + \{K\} \times \bigcap_{I_1, \ldots, I_k \in \mathfrak{R}} T[\![\eta]\!] \ (\psi\,[I_i/\alpha_i, T[\![\tau]\!] \ (\psi\,[I_i/\alpha_i])/\beta]) + \ldots .$$

First, consider the case fst $(v) \neq K$. Then, by definition, $E[\![\texttt{let}\ K\ x = e\ \texttt{in}\ e']\!]\ \rho = \perp$, and we are done, since $\perp \in T[\![\tau']\!]\ \psi$.

In the more interesting, second case, fst $(v) = K$. Then

$$\mathsf{snd}\ (v) \in \bigcap_{I_1, \ldots, I_k \in \mathfrak{R}} \bigsqcup_{J_1, \ldots, J_n \in \mathfrak{R}} T[\![\hat{\tau}]\!] \ (\psi\,[I_i/\alpha_i, J_j/\gamma_j, T[\![\tau]\!] \ (\psi\,[I_i/\alpha_i])/\beta])$$

Let $\alpha_1, \ldots, \alpha_h$, $h \leq k$, be those variables among $\alpha_1, \ldots, \alpha_k$ that are free in $\hat{\tau}\ [\tau/\beta]$.

We now choose a finite a such that $a \leq \mathsf{snd}\ (v)$, thus

$$a \in \bigcap_{I_1, \ldots, I_h \in \mathfrak{R}} \bigcup_{J_1, \ldots, J_n \in \mathfrak{R}} (T[\![\hat{\tau}\ [\tau/\beta]]\!] \ (\psi\,[I_i/\alpha_i, J_j/\gamma_j]))^{\circ}.$$

By definition of set union and intersection, there exist functions $f_1, \ldots_n \in \mathfrak{I}(V)^h \rightarrow \mathfrak{I}(V)$, such that

$$a \in \bigcap_{I_1, \ldots, I_h \in \mathfrak{R}} (T[\![\hat{\tau}\ [\tau/\beta]]\!] \ (\psi\,[I_i/\alpha_i, f_j(I_1, \ldots, I_h)/\gamma_j]))^{\circ}$$

$$\subseteq \bigcap_{I_1, \ldots, I_h \in \mathfrak{R}} T[\![\hat{\tau}\ [\tau/\beta]]\!] \ (\psi\,[I_i/\alpha_i, f_j(I_1, \ldots, I_h)/\gamma_j])$$

$$= \bigcap_{I_1, \ldots, I_h \in \mathfrak{R}} T[\![\hat{\tau}\ [\kappa_j(\alpha_1, \ldots, \alpha_h)/\gamma_j, \tau/\beta]]\!] \ (\psi\,[I_i/\alpha_i, f_j/\kappa_j])$$

$$= T[\![\forall\alpha_1 \ldots \alpha_h.\hat{\tau}\,[\kappa_j(\alpha_1, \ldots, \alpha_h)/\gamma_j, \tau/\beta]]\!] \ (\psi\,[f_j/\kappa_j])$$

$$= T[\![\mathit{gen}\ (A, \mathit{skolem}\ (A, \eta\,[\tau/\beta]))]\!] \ (\psi\,[f_j/\kappa_j]),$$

assuming that the κ_j's are the ones generated by *skolem* $(A, \eta\,[\tau/\beta])$.

Since by definition of *skolem*, none of the κ_j's are free in A, $\models_{\rho, \psi[f_j/\kappa_j]} A$ holds and we can extend A and ρ, obtaining $\models_{\rho[a/x], \psi[f_j/\kappa_j]} A\,[\mathit{gen}\ (A, \mathit{skolem}\ (A, \eta\,[\tau/\beta]))/x]$.

We now apply the inductive assumption to the last premise,

$$A\,[\mathit{gen}\ (A, \mathit{skolem}\ (A, \eta\,[\tau/\beta]))/x] \vdash e' : \tau',$$

and obtain

$$E[\![e']\!]\ (\rho\,[a/x]) \in T[\![\tau']\!] \ (\psi\,[f_j/\kappa_j]) = T[\![\tau']\!]\ \psi,$$

since $FS(\tau') \subseteq FS(A)$. Finally,

$$E[\![\texttt{let}\ K\ x = e\ \texttt{in}\ e']\!]\ \rho = E[\![e']\!] \ (\rho\,[\mathsf{snd}\ (E[\![e]\!]\ \rho)/x])$$
$$= \bigsqcup \{E[\![e']\!] \ (\rho\,[a/x]) \mid a \text{ finite and } a \leq \mathsf{snd}\ (E[\![e]\!]\ \rho)\},$$

by the continuity of E. The latter expression is in $T[\![\tau']\!]\ \psi$ by the closure of ideals under limits.

■

Corollary 8 [Semantic Soundness] If $A \vdash e : \tau$, then $E[\![e]\!]\ \rho \neq \mathsf{wrong}$.

Proof: We apply Lemma 5 to the previous theorem.

References

[BM92] K. Bruce and J. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 316–327, January 1992.

[CDDK86] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.

[CL90] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proc. IFIP Working Conference on Programming Concepts and Methods*, pages 466–491, Sea of Galilee, Israel, April 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[HPW91] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.1. Technical Report YALEU/DCS/RR-777, Dept. of Computer Science, Yale University, New Haven, Conn., August 1991.

[Lä92] K. Läufer. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, New York University, Department of Computer Science, 1992. In preparation.

[LM91] X. Leroy and M. Mauny. Dynamics in ML. In *Proc. Functional Programming Languages and Computer Architecture*, pages 406–426. ACM, 1991.

[LO91] K. Läufer and M. Odersky. Type classes are signatures of abstract types. In *Proc. Phoenix Seminar and Workshop on Declarative Programming*, November 1991.

[Mac86] D. MacQueen. Using dependent types to express modular structure. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 277–286. ACM, Jan. 1986.

[MH88] J. Mitchell and R. Harper. The essence of ML. In *Proc. Symp. on Principles of Programming Languages*. ACM, Jan. 1988.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.

[MMM91] J. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1991.

[MP88] J. Mitchell and G. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.

[MPS86] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71, 1986.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Per90] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1990.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12(1):23–41, 1965.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.

Dynamic Typing in Polymorphic Languages

Martín Abadi, Luca Cardelli
Systems Research Center
Digital Equipment Corporation

Benjamin Pierce
LFCS
University of Edinburgh

Didier Rémy
INRIA Rocquencourt

1 Introduction

Dynamic types are sometimes used to palliate deficiencies in languages with static type systems. They can be used instead of polymorphic types, for example, to build heterogeneous lists; they are also exploited to simulate object-oriented techniques safely in languages that lack them, as when emulating methods with procedures. But dynamic types are of independent value, even when polymorphic types and objects are available. They provide a solution to a kind of computational incompleteness inherent to statically-typed languages, offering, for example, storage of persistent data, inter-process communication, type-dependent functions such as `print`, and the `eval` function.

Hence, there are situations in programming where one would like to use dynamic types even in the presence of arbitrarily advanced type features. In this paper we investigate the interplay of dynamic typing with polymorphism. Our study extends earlier work (see [1]) in allowing polymorphism, but keeps the same basic language constructs (`dynamic` and `typecase`) and the same style.

The interaction of polymorphism and dynamic types gives rise to problems in binding type variables. We find that these problems can be more clearly addressed in languages with explicit polymorphism. Even then, we encounter some perplexing difficulties (as indicated in [1]). In particular, there is no unique way to match the type tagging a dynamic value with a `typecase` pattern. Our solution consists in constraining the syntax of `typecase` patterns, providing static guarantees of unique solutions. The examples we have examined so far suggest that our restriction is not an impediment in practice. This solution applies also to languages with abstract data types, and it extends to languages with subtyping.

Drawing from the experience with explicit polymorphism, we consider languages with implicit polymorphism in the ML style. The same ideas can be used, with some interesting twists. In particular, we are led to introduce tuple variables, which stand for tuples of type variables.

In addition to [1], several recent studies have considered languages with dynamic types [14, 17, 23]. The work most relevant to ours is that of Leroy and Mauny, who define and investigate two extensions of ML with dynamic types. We compare their designs to ours in section 6.

Section 2 is a brief review of dynamic typing in simply-typed languages, based on [1]. Section 3 considers the general case of adding dynamic typing to a language with higher-order polymorphism [11]. An algorithmic formulation of the general framework is obtained in section 3.5 by restricting polymorphism to the second order and placing conditions on the patters used in `typecase` expressions. Sections 4 and 5 discuss abstract data types and subtyping, respectively. Section 6 deals with a language with implicit polymorphism.

2 Review

The integration of static and dynamic typing is fairly straightforward for monomorphic languages. The simplest approach introduces a new base type `Dynamic` along with a `dynamic` expression for constructing values of type `Dynamic` and a `typecase` expression for inspecting them. The typechecking rules for these expressions are:

$$\frac{\Gamma \vdash a \in T}{\Gamma \vdash \texttt{dynamic}(a{:}T) \in \texttt{Dynamic}} \quad \text{(Dyn-I)}$$

$$\frac{\Gamma \vdash d \in \texttt{Dynamic} \qquad \Gamma, x{:}P \vdash b \in T \qquad \Gamma \vdash e \in T}{\Gamma \vdash \texttt{typecase } d \texttt{ of } (x{:}P)\, b \texttt{ else } e \in T} \quad \text{(Dyn-E)}$$

The phrases $(x{:}P)$ and `else` are *branch guards*; P is a *pattern*—here, just a monomorphic type; b and c are *branch bodies*. For notational simplicity, we have

considered only typecase expressions with exactly one guarded branch and an else clause; typecase involving several patterns can be seen as syntactic sugar for several nested instances of the single-pattern typecase.

In the intended implementation, the compilation of a dynamic is a pair consisting of a value and its type:

```
compile[dynamic(a:A)] =
   "<'compile[a]', 'grab[A]'>"
```

The double quotes here indicate that the result of compile is a run-time structure; single quotes mark substructures to be built at compile time. The keyword "grab" indicates a metalevel shift: a compile-time data structure or routine is inserted into the run-time value. Because of this, Dynamic is particularly easy to implement in a bootstrapped compiler, where the run-time and compile-time structures coincide.

The typecase construct uses the compiler's typematch routine to compare the tag of a given dynamic to the branch guard:

```
compile[typecase d of (x:A) b else e] =
  "let d = 'compile[d]' in
   if 'grab[typematch]' ('grab[A]') (snd(d))
      then push[x=fst(d)]; 'compile[b]'
      else 'compile[e]'"
```

In languages with subtyping, it is common to use a subtype test in the typecase construct to give a less restrictive matching rule, allowing a tag to be a subtype of the guard type instead of requiring that the two match exactly.

Constructs analogous to dynamic and typecase have been provided in a number of languages, including Simula-67 [2], CLU [18], Cedar/Mesa [16], Amber [3], Modula-2+ [22], Oberon [25], and Modula-3 [9].

These constructs have surprising expressive power; for example, fixpoint operators can be defined at every type already in a simply typed lambda-calculus extended with Dynamic [1]. Important applications of dynamics include persistence and inter-address-space communication. For example, the following primitives provide input and output of a dynamic value from and to a stream:

```
extern ∈ Writer×Dynamic→Unit
intern ∈ Reader→Dynamic
```

Moreover, dynamics can be used to give a type for an eval primitive [12, 20]:

```
eval ∈ Exp→Dynamic
```

We obtain a much more expressive system by allowing typecase guards to contain pattern variables. For example, the following function takes two Dynamic arguments and attempts to apply the contents of the first (after checking that it is of functional type) to the contents of the second:

```
dynApply =
  λ(df:Dynamic) λ(da:Dynamic)
    typecase df of
      {U,V} (f:U→V)
        typecase da of
          {} (a:U)
            dynamic(f(a):V)
        else ...
    else ...
```

Here U and V are pattern variables introduced by the first guard. In this example, if the arguments are:

```
df = dynamic((λ(x:Int)x+2):Int→Int)
da = dynamic(5:Int)
```

then the typecase guards match as follows:

Tag:	Int→Int
Pattern:	U→V
Result:	{U = Int, V = Int}

Tag:	Int
Pattern:	Int
Result:	{}

and the result of dynApply is dynamic(7 : Int).

A similar example is the dynamic-composition function, which accepts two Dynamic arguments and attempts to construct a Dynamic containing their functional composition:

```
dynCompose =
  λ(df:Dynamic) λ(dg:Dynamic)
    typecase df of
      {U,V} (f:U→V)
        typecase dg of
          {T} (g:T→U)
              dynamic(f∘g:T→V)
        else ...
    else ...
```

3 Explicit Polymorphism

This formulation of dynamic types may be carried over almost unchanged to languages based on explicit polymorphism [11, 21]. For example, the following function checks that its argument df contains a polymorphic function f taking elements of any type into Int. It then creates a new polymorphic function that accepts a type and an argument of that type, instantiates f appropriately, applies it, and squares the result:

```
squarePolyFun =
  λ(df:Dynamic)
    typecase df of
      {} (f:∀(Z)Z→Int)
         λ(W) λ(x:W) f[W](x)*f[W](x)
    else λ(W) λ(x:W) 0
```

Here the type abstraction operator is written λ. Type application is written with square brackets. The types of polymorphic functions begin with ∀. For example, ∀(T)T→T is the type of the polymorphic identity function, λ(T) λ(x:T) x.

3.1 Higher-order pattern variables

First-order pattern variables, by themselves, do not give us sufficient expressive power in matching against polymorphic types. For example, we might like to generalize the dynamic application example from section 2 so that it can accept a polymorphic function and instantiate it appropriately before applying it:

```
dynApply2ltd =
  λ(df:Dynamic) λ(da:Dynamic)
     typecase df of
        {} (f:∀(Z)?→?)
           typecase da of
              {W} (a:W))
                  dynamic(f[W](a):?)
           else ...
     else dynApply(df)(da)
```

But there is no single expression we can fill in for the domain of f that will make dynApply2ltd apply to both:

```
df = dynamic
       (λ(Z) λ(x:Z×Z) <snd(x),fst(x)>: ...)
da = dynamic(<3,4>: ...)
```

and:

```
df = dynamic((λ(Z) λ(x:Z→Z) x): ...)
da = dynamic((λ(x:Int) x): ...)
```

Thus we are led to introducing higher-order pattern variables, which range over "pattern contexts"—patterns abstracted with respect to some collection of type variables. These suffice to express polymorphic dynamic application:

```
dynApply2 =
  λ(df:Dynamic) λ(da:Dynamic)
     typecase df of
        {F,G} (f:∀(Z)F(Z)→G(Z))
           typecase da of
              {W} (a:F(W))
                  dynamic(f[W](a):G(W))
           else ...
     else dynApply(df)(da)
```

For example, if:

```
df = dynamic(id:∀(A)A→A)
da = dynamic(3:Int)
```

then the typecase expressions match as follows:

Tag:	∀(A)A→A
Pattern:	∀(Z)F(Z)→G(Z)
Result:	{F = Λ(X)X, G = Λ(X)X}
Tag:	Int
Pattern:	F(W) (which reduces to W)
Result:	{W = Int}

and the result of the application dynApply2(df)(da) is dynamic(id[Int](3) : Int).

3.2 Syntax

We now formalize dynamic types within the context of a higher-order polymorphic λ-calculus, F_ω [11]. The syntax of F_ω with type Dynamic is given in Figure 1.

In examples we also use base types, cartesian products, and labeled records in types and patterns, but we omit these in the formal treatment.

We regard as identical any pair of formulas that differ only in the names of bound variables. For brevity, we sometimes omit kinding declarations of the form ": Type" and empty pattern-variable bindings. Also, it is technically convenient to write the pattern variables bound by a typecase expression as a syntactic part of the pattern, rather than putting them in front of the guard as we have done in the examples. Thus, typecase e1 of {V}(x:T)e2 else e3 should be read formally as typecase e1 of (x:{V:Type}T)e2 else e3.

3.3 Tag Closure

One critical design decision for a programming language with type Dynamic is the question of whether type tags must be closed (except for occurrences of pattern variables), or whether they may mention universally bound type variables from the surrounding context.

In the simplest scenario, dynamic(a:A) is legal only when A is a closed (but possibly polymorphic) type. Similarly, we would require that the guard in a typecase expression be a closed type.

If the closure restriction is not instituted, then types must actually be passed as arguments to polymorphic functions at run time, so that code can be compiled for expressions like:

```
λ(X) λ(x:X) dynamic(<x,x>:X×X).
```

where the type X×X must be generated at run time. For languages such as ML, where type information is not retained at run time, the closure restriction becomes essential (see section 6). For now, we consider the general case where tags may contain free type variables.

3.4 Definiteness

The most simple-minded formulation of higher-order pattern variables may seem to provide adequate expressive power, but it is not sufficiently constrained to lead to a viable language design. The problem is that there is no guarantee of unique matches of patterns against tags. For example, when the pattern F(Int) is matched

K	::=	**Type**	kind of types
	|	$K \rightarrow K$	kinds of type operators
T	::=	A	type variables
	|	$T \rightarrow T$	function types
	|	$\forall(A : K)T$	quantified types
	|	$\Lambda(A : K)T$	type operators
	|	$T(T)$	application of a type operator
	|	**Dynamic**	the dynamic type
P	::=	$\{V_1 : K_1, \ldots, V_n : K_n\}\, T$	patterns
e	::=	x	variables
	|	$\lambda(x : T)e$	abstraction
	|	$e(e)$	application
	|	$\lambda(A : K)e$	type abstraction
	|	$e[T]$	type application
	|	**dynamic**$(e : T)$	tagging
	|	**typecase** e **of** $(x : P)\, e$ **else** e	tag matching

Figure 1: Syntax of F_ω terms

against the tag `Bool`, the pattern variable `F` is forced to be `Λ(X)Bool`. But when the same pattern is matched against the tag `Int`, we find that `F` can be either `Λ(X)X` or `Λ(X)Int`. There is no reasonable way to choose. Worse yet, consider `F(W)` or `F(W→Int)` for a pattern variable `W`.

These problems compel us to introduce restrictions on the form of patterns. We may require that a pattern matches a tag in at most one way at run time, and fail otherwise. But this leads to unpredictable matching failures. Therefore we prefer a stronger condition.

Informally, we want to allow only patterns that match each tag in at most one way. This condition is called *definiteness*. For example, assume that the type variables `A` and `B` and the operator variable `H` are bound in the current context, and consider the following patterns:

Valid: `{V}` `H(V)`

since, at run time, if `V` appears in the expression to which `H` is bound, then matching is the usual for first-order pattern variables; and otherwise matching leaves `V` completely undetermined, and we set it to a new type constant. Patterns of this form are used in many of our examples, so we want to consider them definite even though they may sometimes leave `V` unconstrained.

Invalid: `{F}` `F(A)`

because, for example, it can match the tag `A→A` in four ways, instantiating `F` to any of: `Λ(C)C→C`, `Λ(C)C→A`, `Λ(C)A→C`, `Λ(C)A→A`.

Valid: `{F}` `∀(X)F(X)`

since, when the scope of `X` is narrower than that of `F`, we can only match the tag `∀(Y)Y→Y` by instantiating `F` to `Λ(C)C→C`.

Valid: `{F}` `(∀(X)F(X))→F(A)`

since the first occurrence of `F` determines its value.

Valid: `{F,V}` `(∀(X)F(X))→F(V)`

since `F` can be matched first and then considered "already bound" at the defining occurrence of `V`.

Invalid: `{F,G}` `∀(X)F(G(X))→G(F(X))`

since neither `F` nor `G` can be considered "already bound" unless the other is bound first.

Note that definiteness of patterns cannot be checked locally. For example,

Valid: `{F}` `F(Int)→F(Bool)`

is definite, although neither occurrence of `F` would be definite in isolation.

The definiteness condition can be formalized, and then one can put a definiteness requirement in the typechecking rule for **typecase**, so that only programs with definite patterns are legal.

Unfortunately, the notion of definiteness does not suggest a typechecking algorithm in any straightforward way. A related problem is that we have no algorithm for the run-time operation of matching patterns against tags. Indeed, it is not known whether the general case of higher-order matching is decidable. Even the second- and third-order cases, whose decidability has been established [15, 7, 8], lead to algorithms too inefficient to be of practical use in implementing **typecase**.

3.5 Second-order polymorphism

To obtain a practical language design, we need a restriction of our general treatment for which efficient typechecking and matching algorithms can be given. We begin by considering the fragment of F_ω with only second-order polymorphism. This restriction is mostly a matter of convenience, and it seems possible that the approach described below applies to the full F_ω.

The syntax of System F (the second-order polymorphic lambda-calculus) with `Dynamic` is given by the following restriction of F_ω with `Dynamic`. We show only the lines that differ.

K	::=	...	
		$\mid$ Type$\rightarrow K$	type operators
T	::=	...	
		$\mid \forall(A)T$	quantified types
		$\mid \Lambda(A)T$	type operators
		$\mid$ ...	
P	::=	...	
e	::=	...	
		$\mid \lambda(A)e$	type abstraction
		$\mid$ ...	

Here, kinds other than `Type` are used only to specify the functionality of pattern variables; abstractions and applications at the level of types are used only to describe patterns. Since every kind has the form

$$\underbrace{\texttt{Type}\rightarrow\cdots\texttt{Type}}_{n},$$

we can simply say that a pattern variable with this kind has *arity n*.

A pattern $\{V_1 : K_1, \ldots, V_n : K_n\}\ T$ is *ordered* if there is some total ordering $<$ of the pattern variables $V_1, \ldots, V_n$ such that each V_i has a *defining occurrence*, that is, a subterm occurrence $U \equiv V_i\, A_1 \cdots A_p$ in T such that:

1. U does not appear in an argument to a pattern variable V_j where $V_j \geq V_i$ (i.e., there is no occurrence $U' \equiv V_j\, Q$ with U a proper subphrase of U' and $V_j \geq V_i$);
2. V_i is fully applied (i.e., the arity of V_i is p);
3. the A_j's are pairwise distinct; and
4. all of the A_j's have narrower scope than V_i (i.e., $A_j \notin \mathrm{FV}(T)$).

Note that this condition can be checked statically.

Ordered matching is a matching algorithm for ordered patterns; given an ordered pattern, this algorithm instantiates variables according to one of the orders that make the pattern ordered. We believe that ordered patterns are always definite, and that ordered matching is correct, that is, it terminates on every input and always yields the same solution independently of the order of variable instantiations. Hence we replace the definiteness condition with the ordered condition in typechecking, and in evaluation we adopt ordered matching. We omit a detailed description of ordered matching; section 6.3 contains a similar algorithm in a somewhat different context.

4 Abstract Types

The interaction between the use of `Dynamic` and abstract data types gives rise to a puzzling design issue: should the type tag of a dynamically typed value containing an element of an abstract type be matched abstractly or concretely? There are good arguments for both choices:

- Abstract matching protects the identity of "hidden" representation types and prevents accidental matches in cases where several abstract types happen to have the same representation.
- Transparent matching allows a more permissive style of programming, where a dynamically typed value of some abstract type is considered to be a value of a different version of "the same" abstract type. This flexibility is critical in many situations. For example, a program may create disk files containing dynamic values, which should remain usable even after the program is recompiled, or two programs on different machines may want to exchange abstract data in the form of dynamically typed values.

By viewing abstract types formally as existential types [19], we can see exactly where the difference between these two solutions lies and suggest a generalization of existential types that supports both. (Existential types can in turn be coded using universal types; with this coding, our design for dynamic types of the previous sections yields the second solution.)

To add existential types to the variant of F_ω defined in the previous section, we extend the syntax of types and terms as in Figure 2.

The typechecking rules for `pack` and `open` are:

$$\frac{S =_\beta \exists(A : K)\ T \qquad \Gamma \vdash e \in [R/A]T}{\Gamma \vdash (\texttt{pack}\ e\ \texttt{as}\ S\ \texttt{hiding}\ R) \in S} \quad (\textsc{Pack})$$

$$\frac{\Gamma \vdash e \in \exists(A : K)\ S \qquad A \notin \mathrm{FV}(T) \qquad \Gamma, A : K, x : S \vdash b \in T}{\Gamma \vdash (\texttt{open}\ e\ \texttt{as}\ [A, x]\ \texttt{in}\ b) \in T} \quad (\textsc{Open})$$

A typical example where an element of an abstract type is packed into a `Dynamic` is:

T	::=	...	
	|	$\exists(A : K)T$	existential types
e	::=	...	
	|	**pack** e **as** T **hiding** T	packing (existential introduction)
	|	**open** e **as** $[A, x]$ **in** e	unpacking (existential elimination)

Figure 2: Extended syntax with existential types

```
let stackpack =
  pack
    push = λ(s:IntList)
             λ(i:Int) cons(i)(s),
     pop = λ(s:IntList) cdr(s),
     top = λ(s:IntList) car(s),
     new = nil
  as Some(X)
       push:X->Int->X,
       pop:X->X, top:X->Int, new:X
  hiding IntList
in
  open stackpack as [Stack,stackops] in
  let dstack =
    dynamic
      (stackops.push(stackops.new)(5):Stack)
  in
     typecase dstack of
       (s:Stack) stackops.top(s)
     else 0
```

Note that this sort of example depends critically on the use of open type tags. As above, open tags must be implemented using run-time types. The evaluation of **pack** must construct a value that carries the representation type.

We have a choice in the evaluation rule for the **open** expression:

- We can evaluate the expression **open** e **as** $[R, x]$ **in** b by replacing the representation type variable R by the actual representation type obtained by evaluating e.
- Alternatively, we can replace R by a new type constant.

Without **Dynamic**, the difference between these rules cannot be detected. But with **Dynamic** we get different behaviors. Since both behaviors are desirable, we may choose to introduce an extended **open** form that provides separate names for the abstract and transparent versions of the representation type:

$$\frac{\Gamma \vdash e \in \exists(H : K)\ S \qquad H \notin \mathrm{FV}(T) \qquad \Gamma, H : K, x : S \vdash [H/R]b \in T}{\Gamma \vdash (\textbf{open } e \textbf{ as } [R, H, x] \textbf{ in } b) \in T} \quad (\textsc{Open})$$

In the body of b, we can build dynamic values with tags R or H; a **typecase** on the former could investigate the representation type while a **typecase** on the latter could not violate the type abstraction.

5 Subtyping

In simple languages with subtyping (e.g., [3, 9]) it is natural to extend **typecase** to perform a subtyping test instead of an exact match. Consider for example:

```
let dx = dynamic(3:Nat)
in
  typecase dx of
      {} (x:Int) ...
  else ...
```

The first **typecase** branch is taken: although the tag of **dx**, **Nat**, is different from **Int**, we have **Nat**≤**Int**.

Unfortunately, this idea runs into difficulties when applied to more complex languages. In general, there does not exist a most general instantiation for pattern variables when a subtype-match is performed. For example, consider the pattern **V→V** and the problem of subtype-matching (**Int→Nat**)≤(**V→V**). Both **Int→Int** and **Nat→Nat** are instances of **V→V** and supertypes of **Int→Nat**, but they are incomparable. Even when the pattern is covariant there may be no most general match. Given a pattern **V** × **V**, there may be a type **A** × **B** such that **A** and **B** have no least upper bound, and so there may be no best instantiation for **V**. This can happen, for example, in a system with bounded quantifiers [6, 10], and in systems where the collection of base types does not form an upper semi-lattice. Linear patterns (where each pattern variable occurs at most once) avoid these problems, but we find linearity too restrictive.

Therefore, we take a different approach that works in general and fits well with the language described in section 3.2. We intend to extend System F with subtyping along the lines of [5]. In order to incorporate also the higher-order pattern variables, we resort to power-kinds [4]. The kind structure of section 3.2 is therefore extended as follows:

K	::=	**Type**	
	|	$K \to K$	
	|	**Power**$(K)(T)$	(where $T : K$)

Informally, **Power(Type)**(T) is the collection of all subtypes of T, and **Power**$(K \to K')(F)$ is the collection of

all operators of kind $K \to K'$ that are pointwise in subtype relation with F. Subtyping ($\leq$) is introduced by interpreting:

$A \leq B : K$ as $A : \texttt{Power}(K)(B)$, where $A, B : K$

$F \leq G : (K \to K')$ as $F(X) \leq G(X) : K'$, for all $X : K$, where $F, G : (K \to K')$

The axiomatization of $\texttt{Power}(K)(T)$ [4] is designed to induce the expected subtyping rules. For example, A : $\texttt{Power}(A)$ says that $A \leq A$.

As in section 3.5, we limit kinds to appear only in patterns, although we may allow bounded quantifiers $\forall(X \leq T)\, T'$ since they can be handled easily. Because of power-kinds, we can now write patterns such as:

```
typecase dx of
  {V,W≤(V×V)} (x:W) ...
  (that is: {V:Type, W:Power(Type)(V×V)} (x:W))
else ...
```

Each branch guard is used in typechecking the corresponding branch body. The shape of branch guards is $\{V_1 : P_1, \ldots, V_n : P_n\}(x : P)$ where each V_i may occur in the P_j with $j > i$ and in P. This shape fits within the normal format of typing environments, and hence introduces no difficulties for static typechecking.

Next we consider the dynamic semantics of **typecase** in presence of subtyping. The idea is to preserve the previous notion that **typecase** performs exact type matches at run time. Subtyping is introduced as a sequence of additional constraints to be checked at run time only after matching. These constraints are easily checked because, by the time they are evaluated, all the pattern variables have been fully instantiated (perhaps to undetermined types, as discussed in section 3). In the example above, suppose that the tag of dx is $(\texttt{Nat} \times \texttt{Int}) \times \texttt{Int}$; then we have the instantiations $\texttt{W} = \texttt{Nat} \times \texttt{Int}$ and $\texttt{V} = \texttt{Int}$. When the matching is completed, we successfully check that $\texttt{W} \leq (\texttt{V} \times \texttt{V})$.

Some examples will illustrate the additional flexibility obtained with subtyping. First we show how to emulate simple monomorphic languages with subtyping but without pattern variables, where **typecase** performs a subtype test. The first example of this section can be reformulated as:

```
typecase dx of
  {V≤Int} (x:V) f[V](x)
else ...
```

where $\texttt{f}:\forall(\texttt{W} \leq \texttt{Int})\texttt{W} \to \texttt{Int}$. In this example, the tag of dx can be any subtype of Int. Note that the assumption $\texttt{V} \leq \texttt{Int}$ is used statically to typecheck $\texttt{f[V](x)}$.

The next example is similar to **dynApply** in section 2, but the type of the argument can be any subtype of the domain of the function:

```
typecase df of
  {V,W} (f:V→W)
    typecase da of
      {V'≤V} (a:V')
        dynamic(f(a):W)
    else ...
else ...
```

With polymorphic tag types, or with polymorphic pattern types with only first-order pattern variables, nothing new happens except that the matching and subtype tests must be the adequate ones for polymorphism.

The next degree of complexity is introduced by higher-order pattern variables. Just as we had $\texttt{V}' \leq \texttt{V}$, a subtype constraint between two first-order pattern variables, we may have $\texttt{F} \leq \texttt{G}:(\texttt{K} \to \texttt{K}')$ for two higher-order pattern variables $\texttt{F},\texttt{G}:(\texttt{K} \to \texttt{K}')$. As mentioned above, the inclusion is intended pointwise: $\texttt{F} \leq \texttt{G}$ iff $\texttt{F(X)} \leq \texttt{G(X)}:\texttt{K}'$ under the assumption $\texttt{X}:\texttt{K}$.

Another form of dynamic application provides an example of higher-order matches with subtyping:

```
typecase df of
  {F,G:Type→Type,V} (f:∀(Z≤V)F(Z)→G(Z))
    typecase da of
      {W≤V} (a:F(W))
        dynamic(f[W](a):G(W))
    else ...
else...
```

Finally, dynamic composition calls for a constraint of the form $\texttt{G}' \leq \texttt{G}$:

```
typecase df of
  {G,H:Type→Type} (f:∀(X)G(X)→H(X))
    typecase dg of
      {F:Type→Type,G'≤G:(Type→Type)}
                          (g:∀(Y)F(Y)→G'(Y))
        dynamic
          ((λ(Z) f[Z]og[Z]):∀(Z)F(Z)→H(Z))
    else ...
else ...
```

This example generalizes to functions of bounded polymorphic types, such as $\forall(\texttt{X} \leq \texttt{A})\texttt{G(X)} \to \texttt{H(X)}$.

6 Implicit Polymorphism

In this section we investigate dynamics in an implicitly typed language, namely ML. First we show that the general treatment of dynamics for explicitly typed languages directly applies to ML, providing explicitly tagged dynamics in an untyped language. This solution is not in the spirit of ML, and all the rest of the section will be devoted to the study of implicitly tagged dynamics in ML.

In the obvious extension of ML, types can still be inferred for all constructs but dynamics; the user needs

to provide type information when creating or reading dynamics. For instance, let us consider the program:

```
twice = dynamic
  (λ(f) λ(x) f(f x):∀(A)(A→A)→(A→A))
```

First, the type scheme ∀(A)(A→A)→(A→A) is inferred for λ(f) λ(x) f(f x) as if it were to be let-bound. Then we check that this type scheme has no free variables and is more general than the required tag ∀(A)(A→A)→(A→A). Conversely, when the extraction of a value from a dynamic succeeds, it is given the type scheme of its tag as if it had been let-bound. Thus, all instances of the value can be used with different instances of the tag as in

```
foo = λ(df)
  typecase df of
     (f:∀(A)(A→A)→(A→A)) <f succ, f not>
  else ...
```

where succ and not are the successor function on integers and the negation function on booleans.

This works perfectly. However, it requires explicit type information in dynamic patterns, which is not in the spirit of ML. Since the ML typechecker can infer most general types for expressions, one would expect the compiler to tag dynamic values with their principal types. For instance, the user writes

```
twice = dynamic(λ(f) λ(x) f (f x))
```

and the dynamic is tagged with ∀(A)(A→A)→(A→A).

However, there is a difficulty with the program

```
apply = dynamic(λ(f) λ(x) f x)
```

Should the dynamic's tag be ∀(A,B)(A→B)→(A→B) or ∀(B,A)(A→B)→(A→B)? As typecase is defined, it succeeds if the tag exactly matches the pattern, including quantifiers. With implicit tagging, the order of quantifiers should not matter.

Moreover, since the tag of twice is more general than the pattern of the function foo, an ML programmer would probably expect that twice can be passed to foo. This is also justified by the fact that the typechecker could have built a dynamic with a weaker tag, and the typecase would have succeeded. That is, in ML, the typecase would be expected to succeed if an instance of the tag matches the pattern. This principle is called *tag instantiation*. Dynamics with tag instantiation but no pattern variables have been implemented in the language CAML [24]. The dynamics studied by Leroy and Mauny [17] have tag instantiation and first-order pattern variables. First-order pattern variables are not powerful enough to type some reasonable examples, such as the applyTwice function shown later. Below we describe a version of dynamics for ML with tag instantiation and second-order pattern variables.

6.1 Tuple variables

Tag instantiation and second-order pattern variables do not fit well together. The difficulty comes from the merging of two features:

- As in the pattern {F} (f:∀(A)F(A)→A), second-order pattern variables may depend on universal variables.
- Tag instantiation requires that if a typecase succeeds, then it also succeeds for a dynamic with an argument that has a more general tag. The tag ∀(A)(A×A)→A matches the previous pattern. So should the tag ∀(A,B)(A×B)→A. But F is not supposed to depend on B!

Because of tag instantiation, polymorphic pattern variables may always depend on more variables than the ones explicitly mentioned. We capture all variables that appear in the tag but that do not correspond to variables in the pattern into a tuple of variables P. The dependence of pattern variable F on all universal variables is written F(P), even though the exact set of variables in P is not statically known. The tuple variable P will be dynamically instantiated to the tuple of all variables of the tag not matched with variables of the pattern. In particular, if the pattern is F(P;A), then P will never contain A.

For instance, the pattern {F} (f:∀(A)F(A)→A) should be written {F} (f:∀(A)F(P;A)→A), so that tag instantiation is possible. The tag ∀(A)(A×A)→A matches this pattern for an empty tuple. The tag ∀(A,B)(A×B)→A matches it for a one-element tuple, namely (B).

Tuple variables bound in different patterns may be instantiated to tuples with different numbers of variables, as in the example just given. Because of such size considerations, it is not always possible to use a tuple variable as argument to an operator, since it may expect an argument of different size. We use tuple sorts in order to guarantee that type expressions are well formed. Formally, a typecase with explicit information should be written, for instance:

```
{p Tuple,F:p→Type→Type}
    (f:∀(P:p,A:Type)F(P;A)→A)
```

where $F(A_0; A_1, \ldots, A_n)$ stands for a fully applied pattern variable $F(A_0)(A_1)\ldots(A_n)$; this notation reminds that the first argument A_0 is a tuple. The sort variable p is to be bound at run time. However, it is not necessary to write the sorts in programs since a typechecker can easily infer them.

6.2 Description of the language

We assume given a denumerable collection of tuple sorts, written π, π', etc., and a sort Type. Then the

sorts are:

$k ::= \pi \mid \mathtt{Type}$	atomic sorts
$K ::= k \mid k \to K$	sorts

Types are:

$T ::= \mathtt{Dynamic} \mid A \mid T(T) \mid T \to T$	types
$P ::= \{A_1, \ldots, A_n\}T$	patterns
$S ::= \forall(A_1 : K_1, \ldots, A_n : K_n)\ T$	type schemes

In traditional ML style, we have left quantifiers implicit in types and hence in patterns. The formation of types, patterns, and contexts is controlled by judgements of the form:

$\Gamma \vdash T \in K$	type T is of sort K in Γ
$\Gamma \vdash S \in K$	type schema S is of sort K in Γ
$\Gamma \vdash P \in K$	pattern P is of sort K in Γ

where contexts are:

$\Gamma ::= \emptyset$	empty context
$\mid \Gamma, \pi$	tuple sort declaration
$\mid \Gamma, F : K$	symbol declaration

Formation rules ensure that type variables are always bound in the proper context, with a sort consistent with their uses. For instance, we have the rule:

$$\frac{\Gamma \vdash T : k \qquad \Gamma \vdash T' : k \to K}{\Gamma \vdash T'(T) : K} \quad (\textsc{Sort-App})$$

In examples only, we help the reader by using different letters for variables of different sorts: type variables of tuple sorts are written `P` and `Q` and type variables of sort `Type` are written `A`, `B`, etc. We also use `F`, `G`, and `H` for pattern variables.

Patterns are pairs of a set of pattern variables $\{V_1, \ldots, V_n\}$ and a type T. They are well formed if the signature of all V_i's if of the form $\pi \to K$ for the same tuple sort π. The exact rule for pattern formation is:

$$\frac{\begin{array}{c}\pi, A_0, A_1, \ldots, A_n \notin \Gamma \\ \Gamma, \pi, V_1 : \pi \to K_1, \ldots, V_n : \pi \to K_n, \\ A_0 : \pi, A_1 : \mathtt{Type}, \ldots, A_n : \mathtt{Type} \vdash T \in \mathtt{Type}\end{array}}{\begin{array}{l}\Gamma, \pi \vdash \{V_1 : \pi \to K_1, \ldots, V_n : \pi \to K_n\}\ T \\ \qquad \in \forall(A_0 : \pi, A_1 : \mathtt{Type}, \ldots, A_n : \mathtt{Type})\ T\end{array}}$$

In particular, there is exactly one pattern variable of tuple sort per pattern.

Again, we would like to guarantee definiteness of patterns, and we impose the sufficient condition that patterns be ordered. Ordered patterns are those that satisfy the conditions given in section 3.5, and in addition all non-pattern free variables of sort `Type` must appear at least once outside of all pattern variables in the pattern. In the context of ML, our definiteness requirement is reminiscent of the type-explication restriction imposed on signatures in Standard ML (see section 7.7 of [13]).

Pattern variables are bound at the beginning of the `typecase`, and their scope is the `typecase` in which they have been introduced. All other free variables are bound at the beginning of the pattern and their scope is the pattern.

Expressions are:

$$\begin{array}{l} e ::= x \mid \lambda(x)\ e \mid e\ e \\ \quad \mid \mathtt{dynamic}(e) \\ \quad \mid \mathtt{typecase}\ e\ \mathtt{of} \\ \qquad \{V_1 : K_1, \ldots, V_n : K_n\}\ \ (x : P)\ e\ \mathtt{else}\ e \end{array}$$

Type inference Pattern variables behave as local type symbols in ML. Typechecking with local type symbols implies an extension of judgement contexts in order to control the scope of type symbols:

$\Gamma ::= \ldots$	
$\mid \Gamma, x : S$	variable type assignment

The typing judgements are $\Gamma \vdash e : T$.

The "instance" rule of ML becomes:

$$\frac{\begin{array}{c} e : S \in \Gamma \\ \Gamma \vdash T : \mathtt{Type} \qquad T \text{ is an instance of } S \end{array}}{\Gamma \vdash e \in T} \quad (\textsc{Inst})$$

It says that instances have to be well formed in the current context, which prevents us from using local symbols out of their scope.

Since we do not want to carry types at run time, we require that the types of values to become dynamic be closed, and then tags can be statically compiled.

$$\frac{\Gamma \vdash e \in S \qquad S \text{ is closed}}{\Gamma \vdash \mathtt{dynamic}(e) \in \mathtt{Dynamic}} \quad (\textsc{Dyn-I})$$

This rule may destroy the principal typing property of ML. If the principal type of an expression e is S and S is not closed, then typing `dynamic`(e) requires that free variables of S are instantiated by ground types. However, the set of closed instances of a principal type that is not closed does not have a principal element.

We want to avoid such situations, since the nonexistence of a principal type corresponds to an ambiguity concerning the tag that a dynamic value should carry. Therefore, we say that a program is not well typed if it has no principal typing derivation.

Type inference is realized by the same algorithm as in ML but delaying tag-closure checking to the end of typechecking (by gathering free variables of types of dynamic values in a list, for instance). If one of these variables is still free at the end of typechecking, then there exists no principal derivation, and the program is not well typed.

The rule for **typecase** is:

$$\frac{\begin{array}{c}\Gamma \vdash d \in \mathbf{Dynamic} \qquad \Gamma \vdash e' \in B \\ \Gamma, \pi \vdash \{V_1 : K_1, \ldots, V_n : K_n\}T \in S \\ \Gamma, \pi, V_1 : K_1, \ldots, V_n : K_n, x : S \vdash e \in B\end{array}}{\begin{array}{c}\Gamma, P : \pi \vdash \mathbf{typecase}\ d\ \mathbf{of} \\ \{V_1, \ldots, V_n\}\ (x{:}T)\ e\ \mathbf{else}\ e' \in B\end{array}}$$

The other rules of ML are unchanged.

6.3 Evaluation

Compilation is easily decomposed into two phases. The first phase translates ML into a variant, called ML^-, where dynamics are explicitly typed; this translation requires a bit of inference. ML^- differs form ML only in its **dynamic** construct:

$$e ::= \ldots \mid \mathbf{dynamic}(e : S)$$

and its typing rule:

$$\frac{\Gamma \vdash a \in S \qquad S \text{ is closed}}{\Gamma \vdash \mathbf{dynamic}(a : S) \in \mathbf{Dynamic}} \quad (\textsc{Dyn-I})$$

The translation of an ML program e into ML^- is any ML^- program M whose principal type derivation is also a principal type derivation for e. This defines M uniquely (types being equal up to alpha-conversion). The type reconstruction algorithm is a trivial adaptation of the usual type inference algorithm. The semantics of an ML program e is the semantics of its translation into ML^-.

The evaluation rules are mostly standard. The only interesting one is for **typecase**, as it involves new methods for matching and pattern-variable instantiation.

Matching is not quite as usual, since it allows tag instantiation, and it also has to deal with tuple variables. Its inputs are a pattern $\{V_1 : K_1, \ldots, V_n : K_n\}\ T$ and a tag, that is, a closed type $\forall(\alpha_1, \ldots, \alpha_n)\ \tau$. The pattern variables are the V_i's, and the universal variables are the remaining free variables of T. The set of variables that occur in the tag (the α_i's) can increase during tag instantiation. The algorithm returns a substitution μ with domain the pattern variables, such that there exists a substitution μ' with domain the tag variables, and with $\mu'(\tau) = \mu(T)$.

We describe the algorithm as transformations on sets of unification constraints called unificands; the transformations keep unchanged the set of substitutions that satisfy the constraints. The substitutions that we consider can instantiate both pattern and tag variables, but not universal variables.

The metavariable T still stands for any type, and τ stands for a type that does not contain pattern variables. The atomic constraints are pairs, $T \doteq \tau$ or $\tau \doteq T$. The pairs $T \doteq \tau$ and $\tau \doteq T$ are considered equal. A substitution μ is a solution of an atomic constraint if it unifies both sides. In addition, the constant $\bot$ is used to represent failure; it is the atomic constraint with no solution.

In general, a unificand U is an atomic constraint, the conjunction of two unificands $U' \wedge U''$, or the existential unificand $\exists\alpha.U'$. The solution set of $U' \wedge U''$ is the intersection of the solution sets of U' and U''. The solution set of the existential unificand $\exists\alpha.U'$ is the set of solutions of U' restricted to variables distinct from α. We identify unificands up to: commutativity and associativity of conjunction, renaming of variables bound by $\exists$'s, exchange of consecutive $\exists$'s, and removal of vacuous $\exists$'s.

Two unificands U and V are equivalent if they have the same set of solutions. This obviously defines an equivalence relation on unificands, and in fact a congruence.

We reduce the original matching problem to that of finding the solutions of the unificand $\exists \mathrm{FV}(\tau).(T \doteq \tau)$. In order to solve this problem, we now give a list of equivalences between unificands—the unificand on top is always equivalent to the one at the bottom. Tag variables are written α; C and C' are constant symbols, and always occur fully applied; and X is either a universal variable A or a pattern variable V. The set of all variables is $\mathcal{V}$.

- Decomposition

$$\frac{C(T_i) \doteq C(\tau_i)}{\bigwedge(T_i \doteq \tau_i')} \qquad \frac{C(T_i) \doteq C'(\tau_j)}{\bot} \quad C \neq C'$$

- Instantiation

$$\frac{C(T_i) \doteq \alpha}{\exists\alpha_i.\left(\alpha \doteq C(\alpha_i) \wedge \bigwedge(T_i \doteq \alpha_i)\right)}$$

$$\frac{V(A_0; \alpha_1, \ldots, \alpha_n) \doteq \tau}{V \doteq \Lambda(A_0; \alpha_1, \ldots, \alpha_n).\tau}$$

- Propagation

$$\frac{X \doteq \tau \wedge M}{X \doteq \tau \wedge [\tau/X]M}$$

$$\frac{\alpha \doteq \tau}{\bot} \qquad \alpha \notin \mathrm{FV}(\tau), \tau \notin \mathcal{V}$$

- Universal-variable restriction

$$\frac{A_i \doteq \alpha \wedge A_j \doteq \alpha}{\bot} \qquad \frac{A \doteq \tau}{\bot} \qquad \tau \notin \mathcal{V}$$

- Existential simplifications

$$\frac{U \wedge (\exists\alpha.U')}{\exists\alpha.(U \wedge U')} \qquad \alpha \notin \mathrm{FV}(U)$$

$$\frac{\exists\alpha.(\alpha \doteq \tau \wedge U)}{\exists\alpha.U} \qquad \alpha \notin \mathrm{FV}(\tau) \cup \mathrm{FV}(U)$$

- Trivial constraints

$$\frac{\alpha \doteq \alpha \wedge U}{U} \qquad \frac{\bot \wedge U}{\bot}$$

These equivalences can be used as rewriting rules. All rules are oriented from top to bottom; one step of rewriting is the application of exactly one rule; applying the rules in any order always terminates. When successful, this process produces canonical unificands of the form:

$$\exists \alpha_k . \Big(\bigwedge (A_i = \alpha_i) \wedge \bigwedge (V_j = \tau_j) \Big)$$

A unificand that cannot be reduced and that is not yet in canonical form is either $\bot$ or contains a constraint $V(A_0; T_1, \ldots, T_n) \doteq \tau$. The ordered condition on patterns prevents the latter (as the second instantiation rule would apply to one of the constraints). Hence, for ordered patterns, rewriting always produces either a canonical unificand or $\bot$.

Because of the form of the rules, the matching is unitary, and all solutions are equal up to renaming of the α_k's. The unique tuple variable that appears in all the τ_j can be bound to the tuple (α_k), and its size bound to the tuple sort.

6.4 Related work

The work on dynamics most closely related to ours is that of Leroy and Mauny [17]. Our system can be seen as an extension of their system with "mixed quantification."

Instead of introducing a **typecase** statement, Leroy and Mauny merge dynamic elimination with the usual case statement of ML. If we ignore this difference, their dynamic patterns have the form QA where A is a type and Q a list of existentially or universally quantified variables.

For instance,

```
∀(A)∃(F)∀(B)∃(G) (v:_(A,F,B,G))
```

is a pattern of their system. The existentially-quantified variables play the role of our pattern variables. The order of quantifiers determines the dependencies among quantified variables. Thus, the pattern above can be rephrased:

```
∃(F)∃(G)∀(A)∀(B) (v:T(A,F(A),B,G(A,B)))
```

Writing quantifiers in our patterns explicitly (for ease of comparison), the equivalent pattern in our system is:

```
{F,G} (v:∀(P,A,B)T(A,F(P;A),B,G(P;A,B)))
```

With the same approach, in fact, we can translate all of their patterns into equivalent patterns in our system, preserving the intended semantics.

On the other hand, there does not seem to be a translation from our language to theirs. They have no pattern equivalent to our pattern:

```
{F,G} (v:∀(P,A,B)T(A,F(P;A),B,G(P:B)))
```

because the quantifiers in the prefix of their patterns are in linear order, and hence it is not possible to have the "parallel" dependencies of F on A and of G on B. We can obtain a system intermediate between theirs and ours by leaving tuple variables implicit, and there we would rewrite the pattern above:

```
{F,G} (v:∀(A,B)T(A,F(A),B,G(B)))
```

However, we believe that explicit tuple variables are useful, since they allow examples like the `applyTwice` function:

```
let applyTwice =
  λ(df) λ(dxy)
    typecase df of
      {F,F'} (f:F(P)→F'(P))
        typecase dxy of
          {G,H} (x,y:F(G(Q)) × (F(H(Q))))
              f x, f y
        else ...
    else ...
```

This cannot be expressed in our intermediate system, nor in systems with just type quantifiers, such as Leroy and Mauny's.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Studentlitteratur (Lund, Sweden), Bratt Institute Fuer Neues Lerned (Goch, FRG), Chartwell-Bratt Ltd (Kent, England), 1979.

[3] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[4] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988.

[5] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In T. Ito and A. R Meyer, editors, *Theoretical Aspects of Computer Software*, number 526 in Lecture Notes in Computer Science, pages 750–770. Springer-Verlag, September 1991.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[7] Gilles Dowek. A second order pattern matching algorithm in the cube of typed λ-calculi. In *Proceedings of Mathematical Foundation of Computer Science*, volume 520 of *Lecture Notes in Computer Science*, pages 151–160. Springer Verlag, 1991. Also Rapport de Recherche INRIA, 1992.

[8] Gilles Dowek. Third order matching is decidable. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, 1992. To appear.

[9] Greg Nelson (ed.). *Systems Programming in Modula-3*. Prentice Hall, 1991.

[10] Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Università di Pisa, March 1990. Technical report TD-6/90, Dipartimento di Informatica, Università di Pisa.

[11] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[12] Mike Gordon. Adding Eval to ML. Personal communication, circa 1980.

[13] Robert Harper, Robin Milner, and Mads Tofte. *Commentary of Standard ML*. The MIT Press, 1991.

[14] Fritz Henglein. Dynamic typing. In *ESOP*, 1992.

[15] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[16] Butler Lampson. A description of the Cedar language. Technical Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

[17] Xavier Leroy and Michel Mauny. Dynamics in ML. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1991.

[18] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.

[19] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

[20] Alan Mycroft. Dynamic types in ML. Draft article, 1983.

[21] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag Lecture Notes in Computer Science 19.

[22] Paul Rovner. On extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6):46–57, November 1986.

[23] Satish R. Thatte. Quasi-static typing (preliminary report). In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 367–381, 1990.

[24] Pierre Weis, Maria-Virginia Aponte, Alain Laville, Michel Mauny, and Ascander Suárez. The CAML reference manual. Research report 121, INRIA, Rocquencourt, September 1990.

[25] Niklaus Wirth. From Modula to Oberon and the programming language Oberon. Technical Report 82, Institut für Informatik, ETH, Zurich, 1987.

Extensions to Standard ML to Support Transactions

Jeannette M. Wing, Manuel Faehndrich, J. Gregory Morrisett, and Scott Nettles*
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A transaction is a control abstraction that lets programmers treat a sequence of operations as an atomic ("all-or-nothing") unit. This paper describes our progress on on-going work to extend SML with transactions. What is novel about our work on transactions is support for multi-threaded concurrent transactions. We use SML's modules facility to reflect explicitly orthogonal concepts heretofore inseparable in other transaction-based programming languages.

1 Revisiting Transactions

1.1 Separation of concerns

Transactions are a well-known and fundamental control abstraction that arose out of the database community. They have three properties that distinguish them from normal sequential processes: (1) A transaction is a sequence of operations that is performed *atomically* ("all-or-nothing"). If it completes successfully, it *commits*; otherwise, it *aborts*; (2) concurrent transactions are *serializable* (appear to occur one-at-a-time), supporting the principle of isolation; and (3) effects of committed transactions are *persistent* (survive failures). Systems like Tabs [8] and Camelot [3] demonstrate the viability of layering a general-purpose transactional facility on top of an operating system. Languages such as Argus [4] and Avalon/C++ [2] go one step further by providing linguistic support for transactions in the context of a general-purpose programming language. In principle programmers can now use transactions as a unit of encapsulation to structure an application program without regard for how they are implemented at the operating system level.

In practice, however, transactions have yet to be shown useful in general-purpose applications programming. One problem is that state-of-the-art transactional facilities are so tightly integrated that application builders must buy into a facility *in toto*, even if they need only one of its services. For example, the Coda file system [7] was originally built on top of Camelot, which supports distributed,

*This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

concurrent, nested transactions. Coda needs transactions for storing "metadata" (e.g., inodes) about files and directories. Coda is structured such that updates to metadata are guaranteed to occur by only one thread executing at a single-site within a single top-level transaction. Hence Coda needs only single-site, single-threaded, non-nested transactions, but by using Camelot was forced to pay the performance overhead for Camelot's other features.

The Venari Project at CMU is revisiting support for transactions by adopting a "pick-and-choose" approach rather than a "kit-and-kaboodle" approach. Ideally, we want to provide separable modules to support transactional semantics for different settings, e.g., in the absence or presence of concurrency. Programmers are then free to compose those modules supporting only those features of transactions they need for their application.

1.2 Non-traditional applications

A second problem with existing transactional facilities is that they have been designed primarily with applications like electronic banking, airline reservations, and relational databases in mind. Non-traditional applications such as proof support environments, software development environments, and CAD/CAM systems want transactional features, most notably data persistence, but have different performance characteristics. For example, these applications do not manipulate simple database records but rather complex data structures such as proof trees, abstract syntax trees, symbol tables, car engine designs, or VLSI designs. Also, users interact with these data during long-lived "sessions" rather than short-lived transactions; indeed we can view a "session" itself as a sequence of transactions. For example, during a proof session a user might explore one path in a proof tree transactionally; if the path begins looking like a dead-end the user may choose to abort, backing all the way up to the first node in the path or perhaps to some intermediate node along the way. Also, though multiple users may need to share these data, simultaneous access might be less frequent. For example, proof developers might work on independent parts of a proof tree, perhaps each proving auxiliary lemmas of the main theorem; software developers might modify different modules of a large program. Finally, these non-traditional applications typically support different update patterns. Whereas travel agents make frequent updates to airline reservations databases, we do not expect to make updates as frequently to proofs of theorems saved in proof libraries.

The Venari Project's application domain is software development environments; one specific problem we are addressing is searching large libraries, e.g., specification and program libraries, used in the development of software systems. We imagine the scenario in which a user searches a large library for a program module that "satisfies" a particular specification. We might wish to perform each query as a transaction, for example, to guarantee isolation from any concurrent update transaction or to abort the query after the first n modules are returned.

Our effort to support a "pick-and-choose" approach for transactions has the advantage of providing us with a way to take performance measurements on different combinations of our separable modules. We have the potential to do different kinds of performance tuning for the non-traditional applications we hope to support.

1.3 Contributions of this paper

SML's modules system lets us cleanly exemplify our "pick-and-choose" approach. In the case of single-site, single-threaded, nested transactions, we support separately the persistence and undoability prop-

erties of transactions in terms of two different modules; we then compose them to build a module for transactions [6]. We reported on this work at the Pittsburgh 1991 ML workshop. Also, along with others at Carnegie Mellon, we have separately designed and built a threads package for SML/NJ [1]. We reported on this work at the Edinburgh 1990 ML workshop. [1]

This paper reports on our progress for combining our support for threads and that for transactions. We address concurrency, in two ways: making an individual transaction multi-threaded and allowing multiple transactions to run concurrently. To the transaction and database community, our work is novel because it is the first to cast within a programming language a model of computation that supports multi-threaded transactions. To the programming language community, our work is among the few to extend the functional programming paradigm to support a traditionally imperative feature. To the SML community, our application of the modules facility should be of particular interest.

In the rest of this paper, we motivate the desire to keep threads and transactions as separate control abstractions (Section 2), give a snippet of the SML programmer's interface to multi-threaded concurrent transactions (Section 3), describe some details of our design (Section 4), discuss what features of SML helped in our design and implementation effort (Section 5), and close with a list of future work (Section 6).

We emphasize that this paper describes on-going work. We are taking a pragmatic, bottom-up approach; by prototyping individual features (e.g., persistence, undoability, read/write locking, nesting, threads, and transactions) incrementally and then combining them in various ways, we can explore a rich design space. This paper focuses on the hardest combination: threads and transactions. (Examples of other reasonable combinations are "multi-threaded persistence" and "multi-threaded undoability.") Thus, for now we are concerned with providing efficient enough run-time *mechanism* to give systems builders flexibility in deciding *policy*. Although the design described in Section 4 does reflect one particular policy, our runtime mechanism is general enough to support alternate policies. Finally, we have not thought greatly about the "ideal" programming interface to provide the SML end-user, but look forward to designing one in the near future.

2 Keeping Threads and Transactions Separate

In languages like Argus and Avalon, a single thread of control is associated with each transaction. But threads and transactions are orthogonal control abstractions. So, we would like to relax the restriction of identifying threads and transactions by allowing multiple threads of control to execute within, and on behalf of, a single transaction.

Figures 1a and 1b depict the traditional model, where we use a wavy line to denote a thread and a box to denote a transaction; time moves from left to right. Figure 1a shows a single thread executing, first entering a transaction and then leaving successfully (i.e., committing). Figure 1b shows two single-threaded transactions executing concurrently. Figure 1c depicts our new model where multiple threads execute within a single transaction. And finally, Figure 1d depicts multi-threaded concurrent transactions, the "composition" of Figures 1b and 1c. The goal of Venari's version of SML is to support Figure 1d through module composition.

[1] For this paper to be somewhat self-contained, we include the cited Pers, Undo, Trans, and Threads signatures in Appendices A and B.

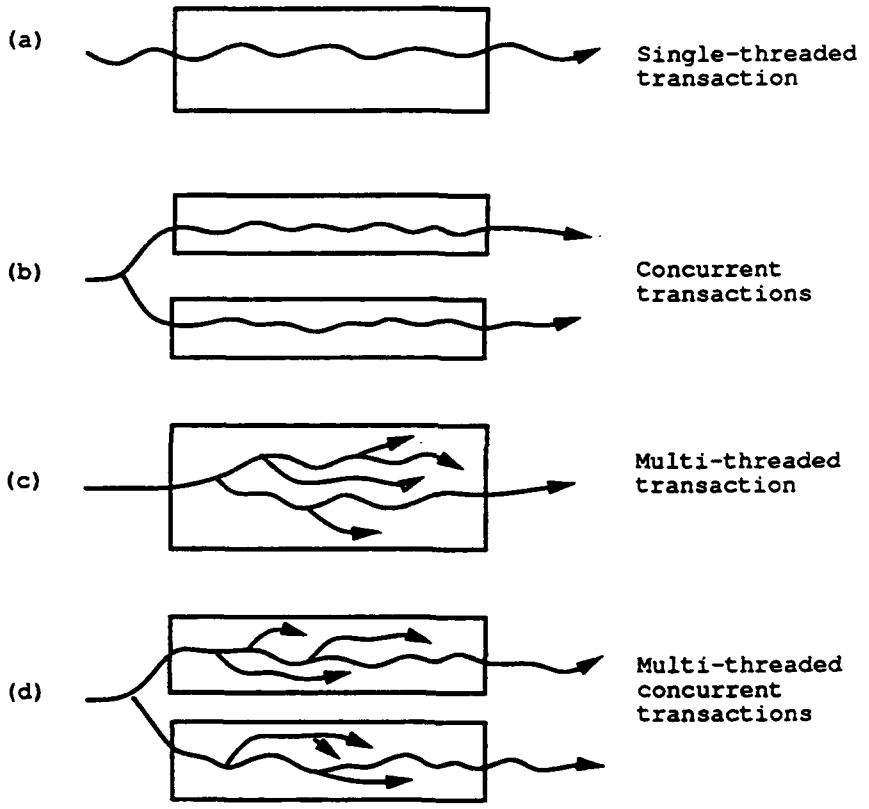


Figure 1: Threads and transactions are separate control abstractions.

2.1 Why have multiple threads within a transaction?

The most compelling argument for supporting multiple threads within a transaction is modularity. Consider the following kinds of multi-threaded programs: (1) a search procedure that uses multiple threads to find program modules satisfying a specification, returning when the first one is found; (2) a procedure with benign side effects, e.g., rebalancing a B-tree or doing garbage collection, that executes in the background of the main computation; (3) a netnews server that uses multiple threads to minimize latency.

We would like to able to run such a multi-threaded program from within a transaction without having to modify the source code. We would like to treat the program as a black box, reuse it in its entirety, but have its effects done transactionally (i.e., atomic, serializable, and persistent). Without being able to simply "wrap" a transaction around the program, we are forced to recode each separate thread as a concurrent subtransaction of a top-level transaction. This violates one aspect of modularity since the entire program has to be recoded.

2.2 Why have multiple threads at all?

Concurrent transactions have to be serializable. Thus, by definition, we can view transactions as happening one after another. On the other hand, threads are often used for two-way communication through shared, mutable resources (e.g., refs). If we identify each thread with a single transaction, then we can no longer do two-way communication between threads. For instance, assuming we associate each thread with a transaction, then Figure 2a shows thread/transaction A and thread/transaction B executing concurrently. Transaction semantics require that the effects of A and B executing concurrently are the same as that of either A executing first followed by B (Figure 2b), or vice versa. Suppose A sends a message to B and B wants to acknowledge A; we cannot put A's execution before B (since A will never get the acknowledgment) nor can we put B before A (since B will never get the message). Thus if we want to support two-way communication between processes, we need to support multiple threads independent of transactions.

Another argument for supporting both threads and transactions as orthogonal concepts is performance. In existing transactional systems, the runtime cost of creating and managing a transaction ("heavyweight" process) is not the same as that for a thread ("lightweight" process). Transactions require runtime mechanism to support protocols for locking, logging, committing/aborting, and crash recovery. There are cases when parallelism is desired without the performance overhead of transactions. Again, even if we were to recode one of our example multi-threaded programs with transactions, we probably do not want to incur the cost of making each thread a transaction.

In short, transactions provide features that threads do not: persistence, undoability, isolation of effects, atomicity of a sequence of operations, and crash recovery. Threads provide functionality, e.g., two-way communication, and performance benefits that transactions do not.

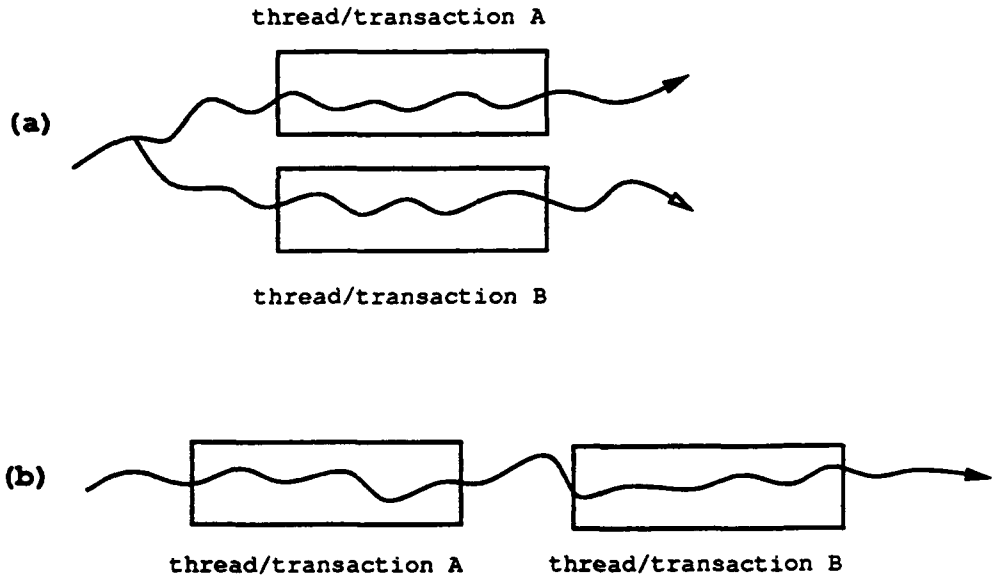


Figure 2: Transactions are serializable.

3 Design Overview

As in our design for single-threaded transactions for SML [6], if `f` is a function applied to some argument `a`, then to execute:

```
f a
```

in a transaction, we want programmers to be able to write:

```
(transact f) a
```

or more probably,

```
((transact f) a ) handle Abort => [some work]
```

Here `f` might be multi-threaded. Informally, the meaning of calling `f` with `transact` is the same as that of just calling `f` with the following additional side effects: If `f` returns normally, then the transaction commits, and if it is a top-level transaction, its effects are saved to persistent memory (i.e., written to disk). If `f` terminates by raising the exception Abort, then the transaction aborts and all of `f`'s effects are undone. Through SML's exceptional handling, in the case of an aborted transaction, the programmer has control of what to do such as clean-up and/or reraising Abort. Note that we support the usual model for subtransactions: the persistence of a child's effects is relative to the commit of its parent and aborting a child does not imply the abort of its parent.

We have implemented the interfaces shown in Figure 3. We use standard two-phase read/write locks to ensure serializability among concurrent transactions. We use Moss's locking rules for nested concurrent transactions [5].

Two items of note are visible through these interfaces. First, the TRANSACT signature shows the clear separation between threads (the substructure Thread_System) and transactions (the substructure Trans).[2] The functor header additionally shows how we have achieved modularization of our support: concurrency within a transaction is packaged in TRANS_THREAD; concurrency among transactions, in RW_LOCK; transaction undoability, in UNDO.

Second, we guarantee the principle of isolation for transactions by making use of "safe" refs [9] (and correspondingly "safe" arrays). In the context of just threads, a normal SML ref is unsafe, while a ref protected by a mutex is a safe ref. In the context of transactions, a ref protected by only a mutex is an unsafe ref, while a ref protected by both a mutex and a read/write lock is a safe ref. A read or write of a safe ref will fail unless the thread (transaction) holds the mutex (read/write lock) of the ref. Thus, it is impossible to violate the isolation principle if the programmer uses only safe refs.

[2] Appendix B shows parts of the THREAD_SYSTEM and other relevant signatures; see [1] for a discussion of threads in SML.

```
signature TRANSACT =
    sig
        structure Trans :
            sig
                exception Abort
                val transact : ('a -> 'b) -> 'a -> 'b
                val abort_top_level: unit -> 'a
                val abort: unit -> 'a

                eqtype rw_lock
                val rw_lock       : unit -> rw_lock
                val acquire_read  : rw_lock -> unit
                val acquire_write : rw_lock -> unit
            end

        structure SRef   : SREF
        structure SArray : SARRAY

        structure Thread_System : THREAD_SYSTEM

        sharing type SRef.lock = SArray.lock = Trans.rw_lock
        sharing type SRef.uref = Thread_System.SRef.sref
        sharing type SArray.uarray = Thread_System.SArray.sarray
    end

functor Transact (structure TT     : TRANS_THREAD
                  structure RW     : RW_LOCK
                  structure SRef   : SREF
                  structure SArray : SARRAY
                  structure Undo   : UNDO
                  sharing type SRef.lock = SArray.lock
                  sharing type SArray.lock = RW.T
                  sharing type SRef.uref = TT.TS.SRef.sref
                  sharing type SArray.uarray = TT.TS.SArray.sarray)
    : TRANSACT = struct ... end
```

Figure 3: Signature and functor modules for transactions.

4 Design Details

4.1 Simplifying assumptions

To simplify our model, and hence our design, we assume that there is exactly one thread that enters a transaction and exactly one that leaves a transaction. We do not lose any generality since we can always immediately fork off multiple children upon entry and we can always force all threads to join into one upon exit. Second, again without loss of generality, we will assume that conceptually the thread exiting is the same as that entering; we call this the transaction's *root thread.*[3] Finally, we assume no mutexes are held before a transaction begins. We make this assumption so we do not have to reacquire locks that were held upon entering a transaction in case an abort occurs. Doing so could cause a deadlock: Suppose the entering thread t holds a lock and then releases it sometime during the transaction. A thread s outside the scope of the transaction then acquires it. If the transaction now aborts, and we are to undo all of its effects, including reacquiring the lock, we may deadlock if s is waiting to acquire some other held lock.

4.2 Per transaction state

Just as there is per thread state [1], we assume there is *per transaction state.* This state includes four pieces of separable information:

- The data objects accessed by the transaction. Since the order of modifications to these data objects is important with respect to abort, we call this information the *(data) undo list.*
- The set of mutex locks held by threads within a transaction. We call this information the *mutex lock set.*
- The set of read and write locks held for the duration of the transaction. We call this information the *read-write lock set.*
- The set of threads running on behalf of the transaction.

The first piece of information (*data state*) is separable from the other three (*synchronization state*) which we need to maintain because of concurrency due to threads.

4.3 Who commits and who aborts?

Our design gives the root thread the privilege of committing the transaction and the responsibility of knowing when it is safe to do so. However, for abort, any thread may encounter a state in which the thread cannot back out of and knowingly wish to cause the abort of the entire transaction; the root thread need not be the only thread to determine that an abort is necessary. Thus, rather than requiring such a thread to communicate with the root thread who could then cause the abort, we permit any thread to cause an abort.

[3] We could relax this assumption by letting threads pass a "baton" among each other, where the baton's flow of control would reflect that of the root thread, but this relaxation is unnecessarily general.

4.4 What happens upon commit?

The effect of a commit is to preserve all data state changes made by the committing transaction. Upon commit, we do the following:

1. Stop all other active threads running on behalf of the transaction so they do not continue to modify state;
2. If the transaction is top-level, throw away the data undo list since we do not need to save the old data values; otherwise, anti-inherit the list to its parent.
3. Release all mutex locks held by non-root threads running on behalf of the transaction.
4. Anti-inherit all read/write locks to its parent [5].
5. If the transaction is top-level, save the state of persistent memory.
6. Exit the transaction and continue processing.

4.5 What happens upon abort?

A transaction may voluntarily abort or be involuntarily aborted (e.g., due to a system crash). Following our semantics for single-threaded `transact` [6], we mask any exception as an abort. Moreover, we treat any unhandled exception as an abort. The effect of an abort is to undo all changes to the data state made by all threads executing on behalf of the transaction. Upon abort, we do the following:

1. Stop all other active threads running on behalf of the transaction so they do not continue to modify state;
2. Follow the undo list backwards, rewriting all old data values.
3. Release all mutex locks held by threads running on behalf of the transaction.
4. Release all read/write locks.

It is critical that we first undo the data values, then release mutexes, and then release read/write locks. Data are protected by mutexes; read/write locks are implemented using them. If we were to release mutexes before undoing all data values, then a thread may modify a data object after the old value gets rewritten. In order to release a read/write lock, we need to be able to acquire other mutexes; if we were not to release mutexes before read/write locks, we could end up in a deadlock situation.

5 Where SML Made a Difference

The SML modules facility is key to our "pick-and-choose" approach. It lets us explicitly reflect the inherent orthogonality of concepts like persistence, undoability, multi-threading, and transactions by letting us define separate structures for each and then functors that compose them. The Transact functor that builds a structure for multi-threaded transactions is one example (Section 3). When we prototyped our implementation for single-threaded, non-concurrent transactions (Appendix A), we also

used a functor parameterized over Pers and Undo structures, which respectively provide persistence and undoability.

We also parameterized the Thread_System structure itself so that the programmer can pick-and-choose among separable support for persistence, undoability, and multi-threading. The Fox Project at CMU, for example, needs only multi-threading; it does not have to use a separate threads module, but rather it just has to apply the Thread_System functor to Undo and Pers structures that are essentially "no-ops." Moreover, it is to SML's credit that modifying the original Thread_System structure to work with Undo required only two lines of additional code: to "turn off undo" while doing a thread operation (e.g., acquiring a mutex) and to "turn it back on" when the operation is completed.

Another way we are able to exploit the modules facility is in code reuse. For example, we use only one functor to implement both kinds of safe refs, that for just threads and that for transactions (q.v., `sharing type SRef.uref = TT.TS.SREf.sref` of Figure 3).

Having first-class functions in SML lets us easily support first-class transactions. This view of transactions is a radical departure from the more traditional view taken by other transaction-based programming languages. Programmers in Camelot, Argus, and Avalon write constructs like `begin_transaction ... end_transaction` to bracket transaction boundaries and cannot treat the compound statement as a value.

We relied on the "mostly functional" nature of SML in our implementation of undoability and persistence. For example, our implementation for undoability keeps a log of all modifications to the store and the old values originally assigned to the modified locations. For traditional imperative languages where modifications to the store would be frequent, maintaining and replaying such logs would be expensive. Such a log is inexpensive to maintain for SML since we can assume mutations are rare.

Though we greatly benefit from SML's static typing, one place where we need dynamic types is in our support for persistence. Our interface allows us to add bindings between identifiers and values to a persistent environment; SML cannot statically determine whether the type of the value returned by a subsequent retrieve (e.g., upon startup of a new SML session or upon crash recovery) on some identifier is the same as the type of the value when it was initially bound.

In summary, SML is a nice vehicle with which to express separable concepts. Though SML may not be the natural language of choice in which to support transactions, it is a natural language of choice for the non-traditional applications of transactions that we have in mind. Many of CMU's projects that involve reasoning about programs, Edinburgh's LEGO "proofs-as-programs" system, Cornell's NuPrl system, and AT&T Bell Labs's proof support environment all use SML as their implementation language. These applications need some, if not all, transactional features like data persistence, concurrency control, checkpointing, backtracking, and crash recovery. We hope to provide these potential users with a set of SML modules that they can use in a "pick-and-choose" fashion.

6 Status and Future Work

We have a working prototype of all the interfaces given in this paper, but much work remains:

- ***Language design***: As mentioned in the introduction we have yet to do a design of an SML end-user's interface for multi-threaded concurrent transactions. We are also still exploring different policies that our mechanisms can support. We may export different end-user interfaces, each reflecting a different policy.

- *Semantics*: We have been negligent in working out any formal semantics for our extensions to SML. Some of the challenges specific to SML include a semantics for undoability and a semantics for the interactions between `callcc` and `transact`; specific to transactions, a model of computation and meaning of correctness for multi-threaded concurrent transactions.
- *Implementation*: After our prototype becomes stabler, we intend to build sample applications and perform experiments to measure the costs of our extensions.

Acknowledgments

We thank the rest of the Venari Group for their discussions: Gene Rollins, Amy Moormann Zaremski, Nick Haines, Darrell Kindred, and Drew Dean. Nick and Darrell, in consultation with Scott Nettles and Greg Morrisett, are now rebuilding the prototype implementation originally built by Greg and Manuel Faehndrich. Drew, who is building a file system in SML, may very well be Venari's first real client.

References

[1] E.C. Cooper and J. Gregory Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, CMU, December 1990.

[2] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, pages 57–69, December 1988.

[3] J. Eppinger, L. Mummert, and A. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[4] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM TOPLAS*, 5(3):382–404, July 1983.

[5] J.E.B. Moss. Nested transactions: An approach to reliable distributed computing. Technical Report MIT/LCS/TR-260, MIT, April 1981.

[6] Scott M. Nettles and J.M. Wing. Persistence + Undoability = Transactions. In *Proc. of HICSS-25*, January 1992.

[7] M. Satyanarayanan et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comp.*, 39(4):447–459, April 1990.

[8] A.Z. Spector et al. Support for distributed transactions in the TABS prototype. *IEEE TSE*, 11(6):520–530, June 1985.

[9] A.P. Tolmach and A.W. Appel. Debugging Standard ML without reverse engineering. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 1–12, 1990.

Appendix A: Persistence and Undoability

Figure 4 illustrates persistence and undoability as orthogonal concepts; we compose them to form single-threaded, non-concurrent, nested transactions. In Figure 4a, as a thread executes, a call to `persist` has the effect of saving all values reachable from a persistent handle to disk. In Figure 4b, a call to `checkpoint` has the effect of "remembering" the store at the point of call; a call to `restore` has the effect of undoing the effects on the store, reverting back to that at the (dynamically) last call to checkpoint. Finally, Figure 4c shows how we compose `checkpoint` with `persist` to give us transaction commit; `checkpoint` with `restore` to give us transaction abort. The signatures for persistence, undoability, and transactions that follow are unfortunately slightly different from that described in [6], but do reflect the current working version of our implementation. The primary difference between the TRANSACT signature here and its analogue in Figure 3 is the absence of an interface for read/write locks, which is not needed in the absence of concurrency.

```
signature PERS = sig
     exception INIT_FAILED
     exception PERSIST_FAILED
     val init: string * string * bool -> unit
     val persist: ('a -> 'b) -> 'a -> 'b

     type identifier
     exception UnboundId
     val bind: identifier * 'a -> unit
     val unbind: identifier -> unit
     val retrieve: identifier -> 'a
  end

signature UNDO =  sig
    exception Restore of exn
    val checkpoint: ('a -> 'b) -> 'a -> 'b
    val restore: exn -> 'a

    val exn2restore: ('a -> 'b) -> 'a -> 'b
    val restore2exn: ('a -> 'b) -> 'a -> 'b
    val restore_on_exn: ('a -> 'b) -> 'a -> 'b
  end

signature TRANSACT = sig
    val transact: ('a -> 'b) -> 'a -> 'b

    exception Abort
    val abort_top_level: unit -> 'a
    val abort: unit -> 'a
  end
```

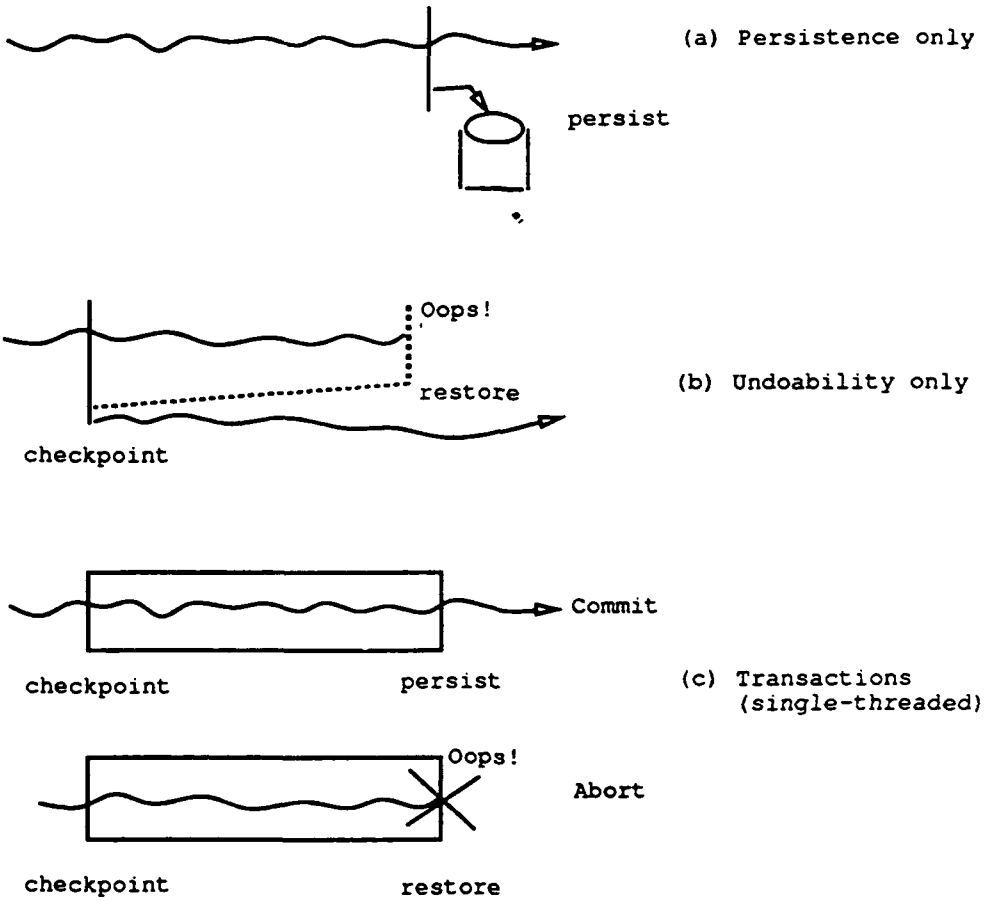


Figure 4: Persistence + Undoability = Transactions

Appendix B: Threads

Our Threads interface and parts of the TRANS_THREAD and THREAD_SYSTEM signatures:

```
signature THREAD = sig
    val fork : (unit -> unit) -> unit
    val exit : unit -> unit

    type mutex
    val mutex : unit -> mutex
    val with_mutex : mutex -> (unit -> 'a) -> 'a

    type condition
    val condition : mutex -> condition
    val with_condition : condition -> (unit -> 'a) -> 'a
    val signal : condition -> unit
    val broadcast : condition -> unit
    val await : condition -> (unit -> bool) -> unit
    val vwait : condition -> (unit -> 'a option) -> 'a

    exception Undefined
    type 'a var
    val var : unit -> '1a var
    val get : 'a var -> 'a
    val set : 'a var -> 'a -> unit
  end

signature TRANS_THREAD = sig
    structure TS : THREAD_SYSTEM
    structure TransID :  sig ... end
    ...
  end

signature THREAD_SYSTEM =  sig
    structure Thread : THREAD

    structure SRef :  sig ... end
    structure SArray :  sig ... end
    ...
  end
```

Programming Images in ML

Emmanuel Chailloux
LIENS*- LITP†

Guy Cousineau
LIENS*

Introduction

In this paper, we describe a library that provides basic types and functions to produce graphical documents in ML. This library is implemented on top of CAML [9] and CAML Light [8] in a functional style [7].

The graphical model is basically that of PostScript [1]. Various objects can be defined on the infinite cartesian plane and arbitrarily scaled, translated and rotated by application of linear transformations. Moreover, high level primitives allow for various combinations of objects.

The main difference with PostScript is that graphic objects are represented by a data structure and has a ML type. A type "picture" is used to represent all printable objects. Pictures have a "frame" and possibly a set of named "handles" that are used for combination operations. Pictures are defined from more basic objects such as geometric elements (polygonal lines, circle arcs and Beziers curves), texts and bitmaps. All operations defined on these types are purely functional except for pixel editing in bitmaps.

Printing is obtained via a translation to PostScript. The philosophy of this translation has been to delegate as much work as possible to PostScript. In particular, the application of linear transformations to pictures is delegated to the PostScript interpretor. This has two advantages: the efficiency of PostScript interpretors is fully used and the sharing involved in the ML representation of pictures is preserved as much as possible.

*URA 1327 - Laboratoire d'Informatique de l' École Normale Supérieure - 45 rue d'Ulm, 75230 Paris Cedex 05, France. Electronic mail: Emmanuel.Chailloux@ens.fr, Guy.Cousineau@ens.fr

†URA 248 - Laboratoire d'Informatique Théorique et Programmation - Institut Blaise Pascal - 4, place Jussieu - UPMC - 75252 Paris Cedex 05, France. Electronic mail : ec@litp.ibp.fr

The main application we foresee for this library is the production of documents involving integrated use of ML and PostScript or ML, LaTeX and PostScript as in the text you are presently reading. In particular, technical texts describing applications written in ML (e.g, proof systems or abstract machines) can be easily decorated with visual representations of ML objects involved in computations (cf. figure 12 showing pictures from Y. Lafont [6] and P. Crégut [4]). Visualization of ML objects can also be used interactively by running an ML window and a PostScript window concurrently. Adding graphical attributes to ML types could lead to interresting debugging tools.

As we all know, functional languages have an expressive power which facilitates the programming activity. It is therefore not surprising that our library provides a way of producing images which is much more pleasant than using the PostScript language itself.

The system we have obtained so far is only a basic layer for producing images in ML. We think it is a good basis for writing technical pictures, described by programs as some of our examples show. We plan to develop further layers to do so.

The paper gives an overview of the system, followed by application examples which reflect some of its possibilities. The last section describes some possible further developments.

1 Overview of the system

1.1 Basic concepts

Each graphic notion has a corresponding ML type. Objects that can be made visible on a graphic device are of type `picture`. A picture always has an absolute position on the infinite cartesian plane. The user can use this fact if it is convenient for her/him or completely forget about it by using more abstract constructions. Pictures can be arbitrarily translated, rotated and scaled. Some pictures can also be created using non-linear transformations (cf. figure 11).

Pictures are produced by grouping more elementary pictures in various ways. Basic constituents of pictures are sketches, texts and bitmaps. A sketch is a sequence of geometric elements such as be lines, arcs and curves. A sketch can be transformed into a picture by choosing a linestyle and a color to draw its constituents or a fillstyle and a color to draw its interior. It can also be used to clip part of a picture. Before being used to produce pictures, sketches can also be arbitrarily grouped and transformed.

1.2 Sketches

A sketch is basically a sequence of geometric elements. It corresponds to what is called a "path" in the PostScript terminology.

The basic constituents of sketches are geometric elements. A geometric element is either a polygonal line represented by a sequence of points or a circle arc represented by a center, a radius and two angles or a Beziers curve represented by a start point, two control points, and an end point. The corresponding CAML types are the following:

```
type point = {xc:float;yc:float};;
type geom_element =
```

```
  Seg of point list
| Arc of point * float * float * float
| Curve of point * point * point * point;;
```

Given the points A=(1,1), B=(1,3), C=(3,3), D=(3,1), E=(1,8), F=(2,4), G=(4,9), H=(5,8) and I=(7,4), the expressions Seg [A;B;C;D;A], Curve(E,F,G,H) and Arc(I,2,30,290) correspond to the the three elements that are drawn with thick lines in figure 1.

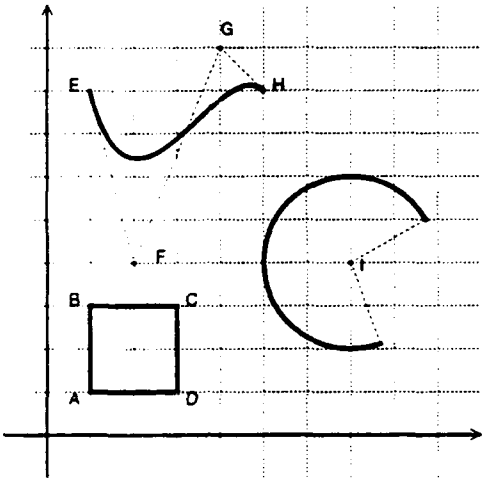


Figure 1: Geometric elements

Sketches are represented by a CAML type **sketch** which is used as an abstract type. The representation of type sketch involves lists of geometric elements together with additional information such as frame and interface information. The basic building function for sketches is:

```
make_sketch : geom_element list -> sketch
```

1.3 Painting Information

Sketches are made of pure lines. To be visualized as images, they require painting information which are represented by the types `linestyle`, `fillstyle` and `color`.

```
type linecap = Buttcap | Squarecap | Roundcap;;
type linejoin = Beveljoin | Roundjoin | Miterjoin;;
type linestyle = {linewidthLS:float; linecapLS:linecap;
                  linejoinLS:linejoin; dashpatternLS:float list};;
type fillstyle = Nzfill | Eofill;;
type color = Rgb of float * float * float
           | Hsb of float * float * float
           | Gra of float;;
```

We do not detail here the meaning of these types which refer exactly to PostScript notions.

1.4 Pictures

The type `picture` is used for all visual objects in the system. Pictures can be built from sketches using painting information. As shown later, they can also be built from bitmaps and texts.

Here are the two functions that make pictures from sketches:

```
make_draw_picture : linestyle * color -> sketch -> picture
make_fill_picture : fillstyle * color -> sketch -> picture
```

Pictures can be "put together" using functions:

```
join_pictures : picture -> picture -> picture
join_picture_list : picture list -> picture
```

The arguments of these functions are taken from left to right Each new picture can cover previous ones partially or totally. Figure 2 shows examples of what can be done using a sketch representing letter P. The left image is a draw picture , the middle one is a fill picture and the right one is a superposition of the two.

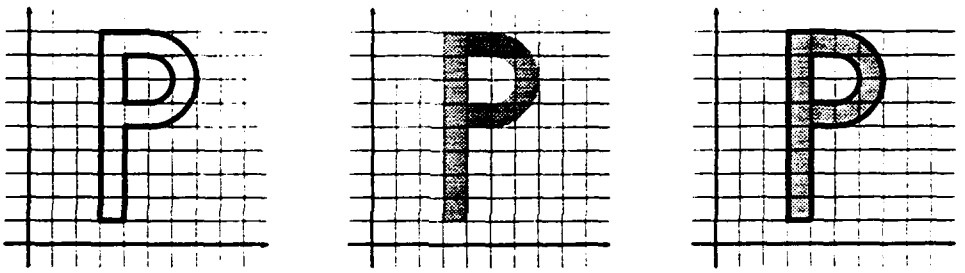


Figure 2: Three pictures obtained from the same sketch

1.5 Picture transformations

Transformations are normally functions of type `point -> point`. However, most useful transformations are linear transformations represented by 3×3 matrix which have shape:

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \textit{operates on vector} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \textit{representing point } (x, y).$$

Users normally build linear transformations using library functions such as:

```
translation : float * float -> transformation
rotation : point -> float -> transformation
scaling : float * float -> transformation
compose_transformation : transformation -> transformation -> transformation
inverse_transformation : transformation -> transformation
```

Transformations are applied to pictures using function

```
transform_picture : transformation -> picture -> picture
```

Figure 3 describes the effect of applying transformations T1= translation (2,-9) , T2= scaling (0.5,0.5) and T3= rotation xc=-2;yc=-2 60 to a basic picture.

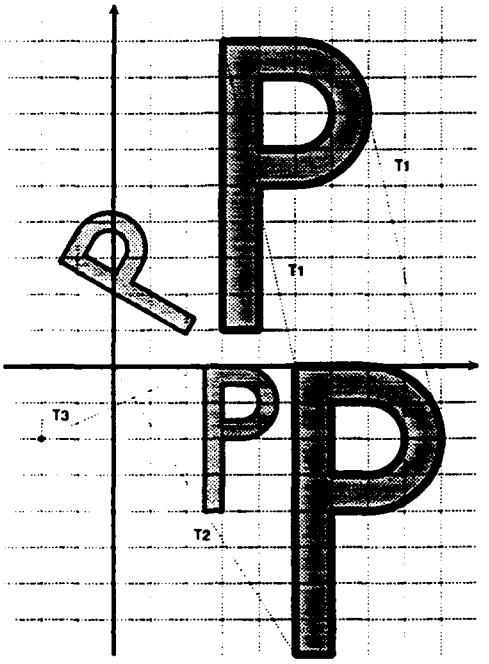


Figure 3: Transformations T1,T2 and T3

1.6 Frames

The CAML graphic system maintains a frame information for pictures. A frame is a rectangle with sides parallel to the axes which contains the picture. It can be made to be the smallest rectangle containing the object but the user can also choose to determine the frame of an object by himself and under his own responsibility for instance to add blank space around an object. Frames provide the basis to define various operations such as rotation of an object around its center (defined as its frame center), horizontal and vertical flip operations, changing the size of an object to fit it in a different frame or putting an object besides or over another object adjusting their frame width and frame height.

Typical functions using frames explicitly are:

```
frame_center : frame -> point
frame_to_frame_transform : frame -> frame -> transformation
picture_frame : picture -> frame
extend_picture_frame : extension -> float -> picture -> picture
```

```
fit_picture_in_frame : picture -> frame -> picture
force_picture_in_frame : frame -> picture -> picture
```

The functions `extend_picture_frame` and `force_picture_in_frame` are used under user's responsibility to give arbitrary frame to object.
The function `fit_picture_in_frame` involves scaling and translating a picture to make it fit exactly in a given frame. The transformation that is used to do so is accessible to the user via function `frame_to_frame_transform`.

The figure 4 shows the result of fitting a given picture in a given frame.

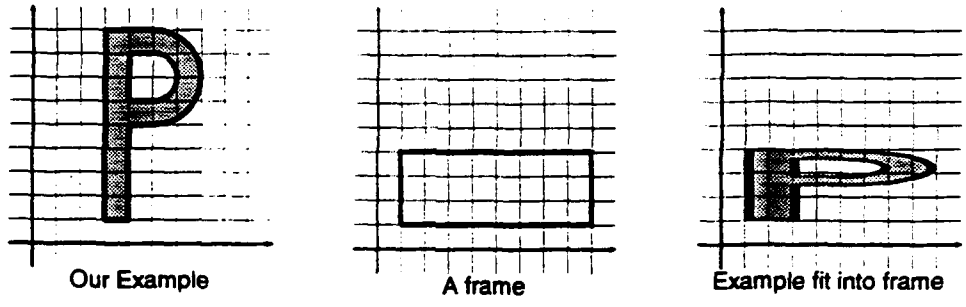


Figure 4: Fitting a picture into a given frame

Some local picture transformations are computed using the frame such as the following ones:

```
rotate_picture : float -> picture -> picture
vflip_picture : picture -> picture
hflip_picture : picture -> picture
```

Figure 5 shows the effect of these functions on our favorite example:

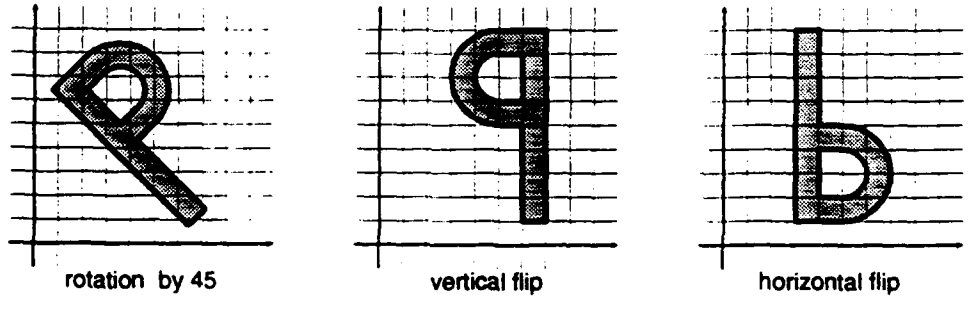


Figure 5: Transformations using frames

1.7 Texts

Text pictures are strings of characters that will be displayed in a given font style and a given size. They are built with their bottom left corner at the origin of the coordinate system but then they can of course be moved to any position.

The basic functions for producing text pictures are:

```
make_font : font_style -> float -> font
make_text_picture : string -> font -> color -> picture
```

An extensive list of font styles exists and the user can have access to the width of any string in any font. Using this information, one can for instance write text on a circle in the following way:

```
#let Pi = acos (-1.0);;
Pi : float
Pi = 3.14159265358979
#let circletext fnt  str center radius =
#  let l= text_width fnt str in
#    if l >= 2.0*Pi*radius
#      then failwith "text is longer than circumference"
#      else let start_angle = (Pi/2.0) + l/(2.0*radius)
#           and char_list = explode str  in
#             let  angle_list = start_angle:: (rev (tl (snd
#                      (it_list (fun (a,l) w -> let aa = a-w/radius
#                                               in  (aa, aa::l))
#                               (start_angle,[])
#                               (map (text_width fnt) char_list)))))
#             and place_char  (s,a) =
#                 let T1 = rotation center ((a-Pi/2.0)*180.0/Pi)
#                 and T2 = translation (center.xc,center.yc+radius)
#                 in transform_picture (T1 CT T2)
#                     (make_text_picture s fnt (Gra 0.0))
#             in join_picture_list
#                  (map place_char (combine (char_list,angle_list)));;
circletext : font -> string -> point -> float -> picture
circletext = <fun>
```

The result of applying `circletext` with center (5,5) and radius 3 is shown in figure 6.

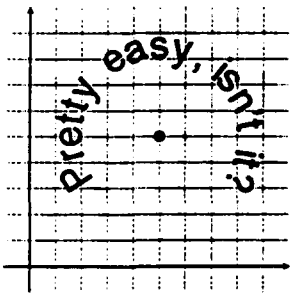


Figure 6: A circular text

1.8 Bitmaps

A bitmap is basically a two dimensional array of pixels. It has a height, a width and a depth. The depth is the number of bits used for each pixel. Possible values are 1,2,4 or 8.

Bitmaps can be created from scratch using functions

```
create_bitmap : int -> int -> int -> bitmap
set_pixel : bitmap -> int -> int -> int -> char
```

However, they are usually read from a string representation obtained through a digitalizer by function:

```
read_bitmap : int -> string -> bitmap
```

whose first argument is the depth.

Functions are provided to uniformly modify a bitmap or extract a sub_bitmap such as

```
map_bitmap : (int -> int) -> bitmap -> bitmap
sub_bitmap : bitmap -> int * int -> int * int -> bitmap
```

Figure 7 shows an example of a bitmap of depth 1 and its inverted image.

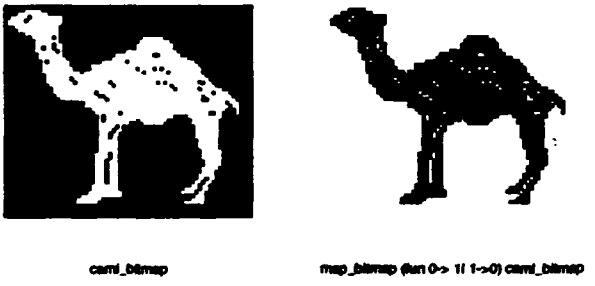


Figure 7: A bitmap transformation

1.9 Picture Composition using frames

Frames can be used to adjust pictures such that their frames fit nicely together either side by side or one on top of the other. Functions `BPICT` and `OPICT` make a new pictures by putting two pictures side by side or one on top of the other. If necessary, the second argument is scaled so that its frame gets same height or width as that of the first argument. Other functions perform similar actions but do not scale the second argument. The two arguments are then aligned in specified ways.

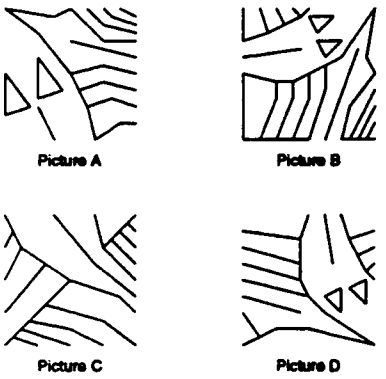
Using these composition functions, the users can forget completely about the cartesian plane in which pictures are built. Only the relative sizes of pictures are relevant.

This technique of picture composition is nicely exemplified by the construction of an Escher picture called "Square limit". A programmed version of this picture has

been given by Henderson in [5]. A detailed account of Henderson's approach is also given in Course Notes by Cousot [3]. The version presented here uses the same basic pictures but builds the final picture in a different way.

The building blocks of the final picture are four basic pictures `A,B,C,D` and three functions named `trio`, `quartet` and `cycle`.

Here are the four basic pictures:

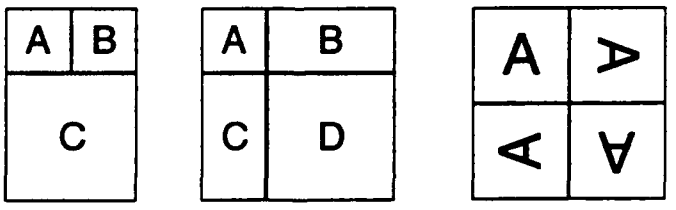


The 4 basic pictures

The three basic functions are defined in the following way:

```
#let rot = rotate_picture 90.0;;
rot : picture -> picture
#let trio(p1,p2,p3) =  (p1 BPICT p2) OPICT p3;;
trio : picture * picture * picture -> picture
#let quartet (p1,p2,p3,p4) = (p1 BPICT p2) OPICT (p3 BPICT p4);;
quartet : picture * picture * picture * picture -> picture
#let cycle p =  (p BPICT (rot (rot (rot p))))
#               OPICT  ((rot p) BPICT (rot(rot p)));;
cycle : picture -> picture
```

Here is a description of what these functions do:



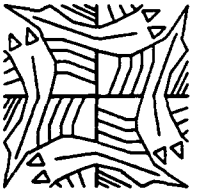
The CAML definition of the picture construction is:

```
let small = scale_picture (0.5,0.5);;
let square_limit n (P,Q,R,S) =
   let TT=quartet(P,Q,R,S)
   and UU=cycle (rot Q)
```

```
in
  let step(C,L,T) =
     (quartet(small C
             ,small(L BPICT T)
             ,small((rot T)OPICT(rot L))
             ,UU)
     ,trio(small L,small T,rot TT)
     ,trio(small L,small T, TT))
  and final_step(C,L,T) =
      quartet(small C,small L, small(rot T),rot Q)
  in
      cycle(final_step(iterate step n (TT,rot TT,UU)));;
```

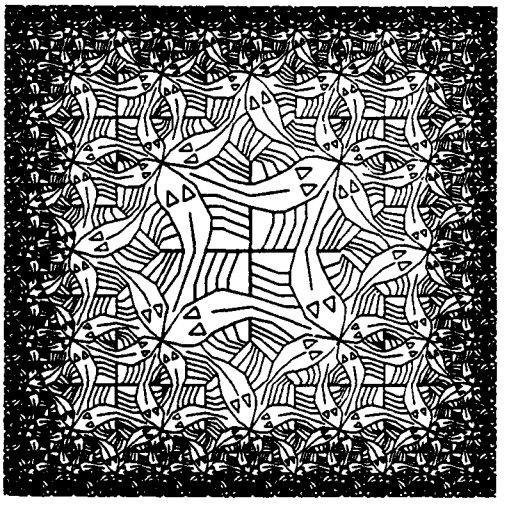
Here are TT and UU for basic picture A,B,C,D:

quartet(A,B,C,D)

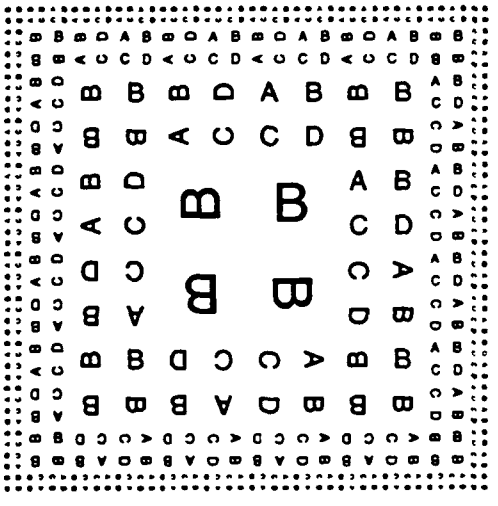


cycle(rot B)

The picture corresponding to the original Escher etching is **square_limit 2 (A,B,C,D)**. It is shown together with its structural description in figure 1.9.



Square Limit



Square Limit structure

1.10 Another form of picture composition

Pictures can be given handles in order to combine them in specific ways to obtain new pictures. A handle is an oriented segment defined by two points. The simplest case is

when pictures have one input handle and one output handle. Pictures can also have sets of named handles in input and output.

The function **APICT** combines two pictures by transforming the second in such a way that the output handle of the first one coincides with the input handle of the second one. The input and output handles of the result are the input handle of the first one and the output handle of the second one.

Figure 8 demonstrates the use of handles. We start with a square bitmap with side equal to one. It has two handles. The input handle is the segment ((0,0),(1,0)) at the bottom of the picture. The output handle is the segment ((1,1),(1,$2-\phi$)) where ϕ is the golden number :$(1+\sqrt{5})/2$. The result is obtained by applying to it the function **gold_spiral** defined by:

```
let rec gold_spiral P = function 0 -> P
                              |   n -> (gold_spiral(n-1)) APICT P;;
```

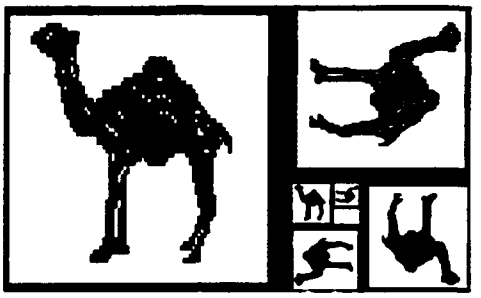


Figure 8: Golden Camels

1.11 Displaying pictures

The display functions produce a PostScript translation for a picture which can be directed to a file. These functions can perform a translation and a uniform scaling of the picture in order to make it centered on the page with reasonable size. So the user does not have to worry about the actual position and size of its image in order to visualize it.

2 Application examples

2.1 Drawing binary trees

Drawing pictures of binary trees that are pleasant to look at requires some computation. The main constraint to satisfy is that subtrees should be not overlap. Another constraint, almost equally important, is that space should be rather uniformly occupied i.e. given two subtrees with the same father, the respective space to allocate to each of them depends on their size and shape. Figure 9 shows two rather different kinds of binary trees.

The design principles we have adopted for drawing binary trees are the followings:

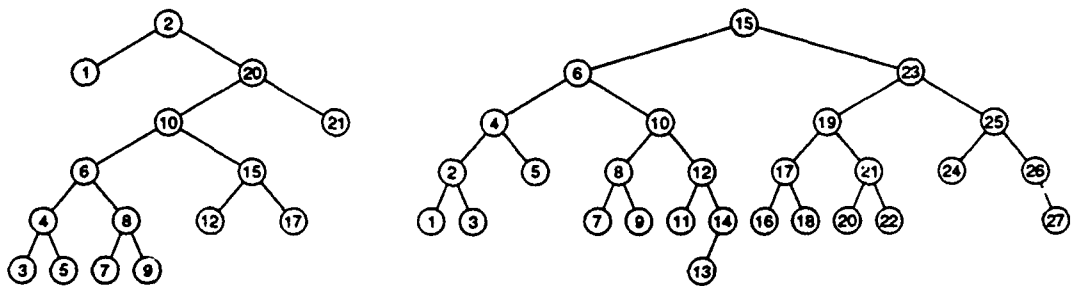


Figure 9: Two examples of binary trees

- At each level of a binary tree, the distance between brother nodes should be constant.
- The distance between brother nodes at level (n+1) should at most equal to the distance at level n.
- Two brother subtrees should be drawn is such a way that at each level, the distance between the rightmost node of the left subtree and the leftmost node of the right subtree should be at least equal to the standard distance between two brother nodes at that level.

These constraints are taken into account by a function `compute_coef_list` which computes for each binary tree a list of coefficients which indicate the ratio that should be adopted between the distance between two brother nodes at level (n+1) and the distance between two brother nodes at level n. Given this list of coefficients, the drawing of a binary tree becomes straightforward. The drawing function is a standard recursive function on binary tree which has as a parameter a function to draw nodes. Different functions can therefore be used to draw nodes. For instance, if trees are AVL trees, it is possible to indicate in each node whether the subtree corresponding to each node is balanced, or heavier on the left or heavier on the right as shown in figure 10.

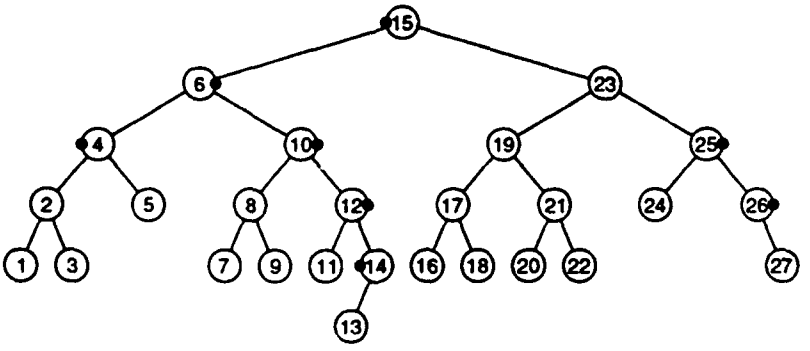


Figure 10: An AVL tree

2.2 Escher's Circle Limit III picture

The picture shown in figure 11 was programmed by students at École Normale Supérieure [2]. It uses knowledge from hyperbolic geometry and group theory.

The circle that contains the picture is the Poincaré representation of the hyperbolic plane. The Escher design is based on a paving of the hyperbolic planes using an isometry group applied to a unique basic picture which is a fish. This group has three generators. By adding explicitly their inverses, it is possible to obtain a canonical rewriting system for checking equality of its elements. It is therefore possible to generate non redundant sets of elements of the group. The picture is obtained using a finite non redundant subset.

The isometries involved correspond in the Poincaré model to homographies of the form:

$$h_{a,\lambda} : z \mapsto \lambda \frac{z+a}{1+\bar{a}z},$$

where a and λ are complex numbers verifying $\mid a \mid < 1$ and $\mid \lambda \mid = 1$.

These non linear transformations have to be computed by ML since PostScript does not know how to handle them. The finite set of fishes that is retained for the final picture are those whose size is greater than a given bound.

The key function is extremely simple. It takes as arguments a minimal bound d for the size and a list of transformations:

```
let rec escher d=
   let ok x =module (app (calc x) q1) < d in
     fun []      -> black_circle
      | (x::l)   -> if  ok x then (escher d l)  JPICT (poisson x)
                    else (escher d l);;
```

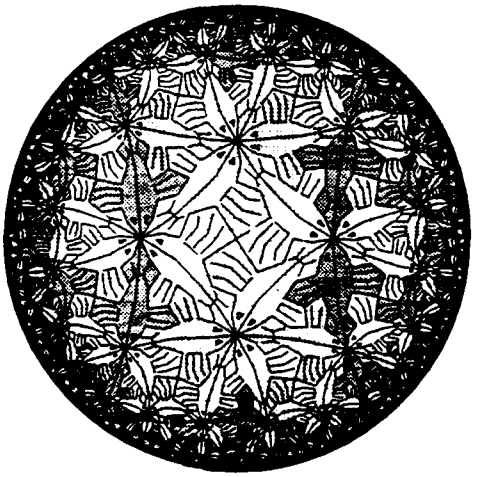


Figure 11: Circle Limit III

2.3 Others contributions

We show two works realized at LIENS by P. Crégut and Y. Lafont (cf. figure 12).

P. Crégut gave a trace in his paper [4], for his abstract machine, to reduce λ-terms. The figure 12 shows the reduction from $\lambda t.(\lambda u.u(\lambda v.u))(\lambda x.x)t)$ to $\lambda t.t(\lambda v.t)$. The conventions used in the drawing are :

- applications are represented by a white circle,
- bound variables by a black circle with the De Bruijn's index,
- free variables by a white circle with the nesting level,
- closures by a white box with the term, its environment, the nesting level and the nesting level of the closure,
- the current state by a black box like a closure with a stack as third term.

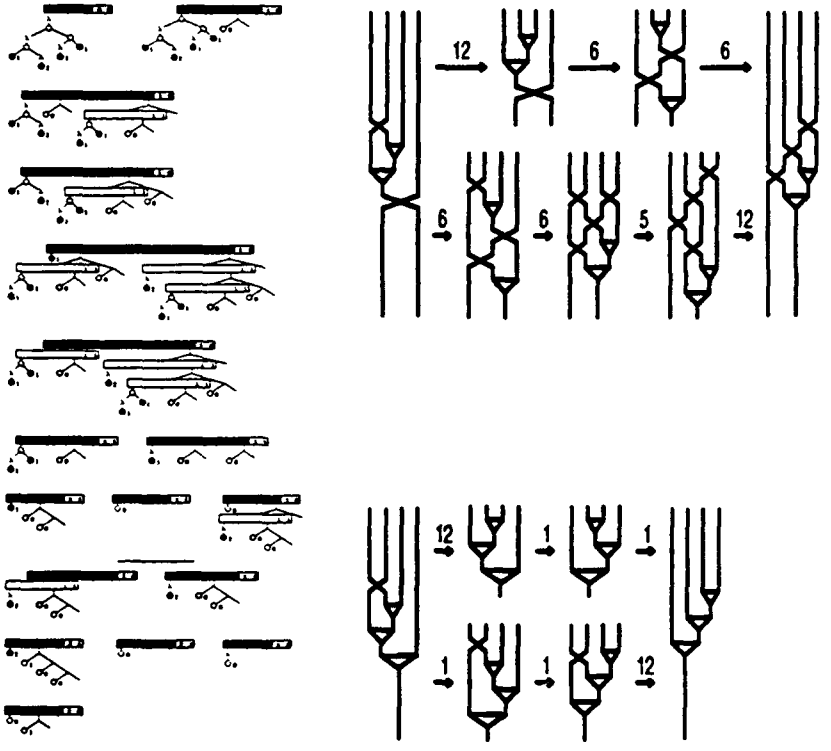


Figure 12: Trace execution and Rewriting rules

For some experiments in 2-dimensional symbolic computation, Y. Lafont is developing a software in the functional programming language CAML using this graphic library. It was used to check the confluence of his twelve rules (only few seconds for checking, a bit more for [illegible]). The fifty-six critical pairs are shown in his paper annex [6] under graphical form.

3 Possible Developments

The library presented here offers the possibility of representing complex images as ML values of type `picture` and gives functions to operate on this type. Basically, it incorporates in ML the possibilities of PostScript with an added value offered by the functional style.

Typical applications are functions of type `user_type -> picture` which give a visual representation for conceptual objects such as for instance binary trees. It is clear that many objects defined by ML users have a visual counterpart that is much more readable and informative than what the standard value printer produces. Associating visualization functions to ML types could be an important debugging aid. It could also greatly facilitate the production of research papers describing systems implemented in ML. The present system is too raw and requires too much effort from the user to produce easily images from ML types. We shall therefore investigate the possibility of building appropriate tools to help him/her in this task. We shall also investigate the possibility of producing informations about the ML system in a graphic way.

Another possibility of application is the production of complex technical pictures. The present situation in this domain is rather frustrating. The production of technical texts has been greatly facilitated by the existence of TEX but no similar facility exists for pictures. We think that a functional description of pictures is a good basis for improving this situation.

References

[1] ADOBE. *PostScript Reference Manual.* Addison-Wesley, 1985.

[2] CHAMBERT-LOIR, A., GRANBOULAN, L., AND LEMAIRE, C. Une œuvre d'Escher en CAML. Tech. rep., "École normale supérieure", 1991. Rapport de projet de Magistère.

[3] COUSOT, P. Cours d'Informatique de l'École Polytechnique. Paris, 1988.

[4] CRÉGUT, P. An Abstract Machine for the Normalization of λ-terms. In *Lisp and Functional Programming* (1990), ACM.

[5] HENDERSON, P. Functional Geometry. In *Symposium on Lisp and Functional Programming* (1982), ACM.

[6] LAFONT, Y. Penrose diagrams and 2-dimensional rewriting. In *Symposium on Applications of categories in Computer Science* (1992), Cambridge University Press, LMS Lecture Notes Series.

[7] LUCAS, P., AND ZILLES, S. Graphics in an Applicative Context. Tech. rep., IBM, Feb. 1987.

[8] MAUNY, M. Functional Programming using CAML Light. Tech. rep., INRIA, Sept. 1991.

[9] WEIS, P., APONTE, M. V., LAVILLE, A., MAUNY, M., AND SUAREZ, A. The CAML reference manual. Tech. Rep. 121, INRIA, Sept. 1990.

Distributed Programming with Asynchonous Ordered Channels in Distributed ML

Robert Cooper[†]
Clifford Krumvieda[‡]
{rcbc,cliff}@cs.cornell.edu

Computer Science Department, Cornell University, Ithaca NY 14853

ABSTRACT

Distributed ML (DML) is an extension of Standard ML for reliable distributed programming. This paper motivates the choice of port groups and asynchronous multicast as DML's distributed communication primitives [Krumvieda 1991].

Multicast is an important tool for parallelism and fault tolerance in distributed programming. It is useful whenever groups of processes must cooperate on some task, or receive the same piece of information. Writing distributed programs comprising many processes is made difficult by the abundance of possible event interleavings caused by concurrency and failures. Using a multicast primitive that provides ordering and reliability guarantees, reduces the possible event histories, making correctness arguments simpler.

Synchronous primitives, such as synchronous send and remote rendezvous, have been popular in the functional language community, because of their simple ordering semantics. But with synchronous communication it is difficult to exploit the full bandwidth of fast computer networks [Birman and van Renesse 1992, Knabe 1991] because each communication necessarily involves one or more round-trip packet exchanges over the network. When multicast is considered, these costs become intolerable. Asynchronous communication permits pipelining for good performance, and uses new ordering properties to substitute for those provided by synchronous primitives.

Distributed ML [Krumvieda 1991] presents flexible, asynchronous group communication facilities using the higher-order concurrency primitives of Concurrent ML [Reppy 1991].

I. INTRODUCTION

Standard ML (SML) [Milner *et al.* 1989] is a modern language that supports the construction of correct programs. For instance, SML provides first-class functions, strong static typing, polymorphism, and exception handling. Several features of the language make it especially appealing to the distributed programmer:

- Most objects in SML are immutable values; unlike C functions, SML applications cannot directly modify complex structures by assigning to their internal fields. Because an immutable

[†] Supported under DARPA/NASA grant NAG-2-593, and by IBM, GTE and Siemens.

[‡] Supported by a National Defense Science and Engineering Graduate Fellowship sponsored by the Air Force Office of Scientific Research/AFSC, United States Air Force, under Contract F49620-86-C-0127.

function parameter can be passed either by value or by reference (both methods produce the same results), a distributed programmer need not worry about semantic differences between local and remote evaluation.

- Many distributed applications evolve from separately running programs that use the network to share common data. These so-called *multi-applications* [Auerbach *et al.* 1991] and their interfaces must be designed well if they are to be combined effectively. SML's module system—which allows module interfaces to be type-checked and parameterized in terms of each other—provides a good framework for designing multi-applications.

- Standard ML has a particularly efficient implementation, Standard ML of New Jersey [Appel and MacQueen 1991], whose code seems to run within a factor of two of optimized C code. Since software costs constitute a significant fraction of the total costs of distributed communication, the low-level support system of a distributed language must be efficient.

- Any general purpose distributed programming system must provide support for node-level (lightweight) concurrency. Concurrent ML (CML) [Reppy 1991], an extension of SML, adds high-level support for single-environment concurrent programming. CML is especially appealing because it provides excellent performance.

At least two research groups have implemented distributed computing extensions to SML [Knabe 1991, Matthews 1991]. Both groups have chosen to implement *synchronous* communication, where data sources (senders) rendezvous with data sinks (receivers) to transfer data. Synchronization between senders and receivers ensures the stability and ordering of message delivery, thereby aiding program correctness. Unfortunately, synchronous communication makes it difficult to exploit the full bandwidth of fast computer networks [Birman and van Renesse 1992]. In Distributed ML (DML) [Krumvieda 1991], senders do not wait for receivers to rendezvous with them, but instead use *asynchronous* communication. We will argue that asynchronous communication primitives are better than their synchronous counterparts for distributed programming languages and explain how the stability and ordering problems are addressed.

Although point-to-point (single sender, single reciever) communication in a distributed system is important, distributed applications frequently employ one-to-many (*multicast*) communication, *e.g.*, to efficiently disperse data and maintain replicated data sets. Multicast can be more difficult to use than point-to-point communication because of the larger number of possible event orderings. But by adding some reliability and ordering properties [Birman *et al.* 1991b], the possible orderings can be substantially reduced and correctness arguments simplified. We will argue for reliable, ordered multicast as a primitive in a distributed programming language.

II. DISTRIBUTED ML

A DML program comprises a set of *nodes* which contain multiple pre-emptive threads sharing access to a heap. Nodes may fail by crashing.

Nodes communicate with each other via *port groups*, collections of `src_ports` (source ports) and `dest_ports` (destination ports) used to multicast data (see Figure 1). A port group serves the same purpose in DML that a channel does in CML: it transmits typed data between threads. Conceptually, data which are placed on a group's `src_port` are transported to each of the group's `dest_ports`.

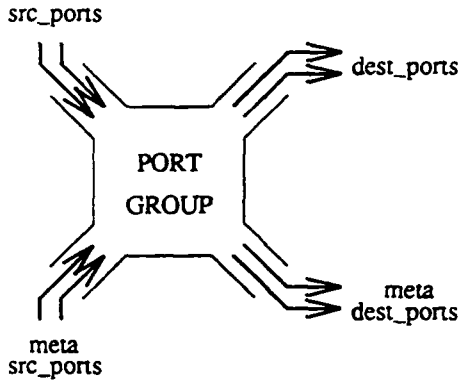


Fig. 1. Conceptual view of a port group

Groups have `meta_src_ports` and `meta_dest_ports` which convey information about port group membership. For instance, the information that a certain `src_port` has failed appears on the group's `meta_dest_port`. Meta ports are examined in detail in a companion paper [Krumvieda 1992].

An abbreviated signature for ports and port groups appears in Figure 2. The `portTransmit` function is used to asynchronously transmit a datum through a `src_port`. It returns a `CML.event` value, which can be used to obtain a message delivery guarantee (see Section V). Data sent through port groups are sent asynchronously, and destination ports act as data buffers.

Port groups with more than one `src_port` must interleave data that are multicast to their `dest_ports`. The programmer specifies one of the orderings defined in the `ORDERINGS` signature to `mkGrp` (discussed in Section IV) when creating the group.

The `portReceive` function is used to receive a value from a destination port. It returns an `OrdEvt.ord_event` value that encodes multicast ordering information. How ordered events differ from CML events is discussed in Section VI.

```
signature PORTS =
  sig
    structure CML: CONCUR_ML
    structure OrdEvt: ORD_EVENT

    type ('a, 'b) src_port
    type 'a dest_port
    type port_id

    val portTransmit: ('a, 'b) src_port * 'a -> 'b CML.event
    val portReceive: 'a dest_port -> 'a OrdEvt.ord_event

    val srcID: ('a, 'b) src_port -> port_id
    val destID: ('a, 'b) dest_port -> port_id
    ...
  end

signature PORT_GROUP =
  sig
    structure Ports: PORTS
    structure Ord: ORDERINGS

    type ('a, 'b) port_group   (* type 'a data and type 'b meta_data *)
    type 'a gview

    val mkGrp: Ord.ordering -> 'la port_group
    val mkSrc: ('a, 'b) port_group * 'b ->
                              ('a, 'b gview) Ports.src_port * 'b gview
    val mkDest: ('a, 'b) port_group * 'b -> 'a Ports.dest_port * 'b gview
    ...
  end
```

Fig. 2. Port and port group signatures

III. MULTICASTS

Just as point-to-point message primitives benefit from properties such as reliability (*e.g.*, retransmitting messages lost by the network) and FIFO ordering, multicast communication is easier to use and reason about if it possesses failure and ordering properties [Birman *et al.* 1991a]. Two of these properties are listed below.

- *Multicast atomicity*: a multicast should be delivered to either all or none of its destinations. A multicast might not be delivered if its sender crashes part way through its transmission.
- *Multicast ordering*: a multicast may be ordered relative to other multicasts in the system. The three properties below generalize the FIFO property common in point-to-point commu-

nication.

- *FIFO ordering* requires multicasts to be ordered pair-wise between the sender and each receiver.
- *causal ordering* preserves potential causality between sends and receives, and is discussed in detail in Section IV.
- *total ordering* ensures that all multicasts are delivered in the same order at all destinations. It enforces an ordering on concurrent multicasts initiated by different senders, and also observes the causal ordering property.

Correctly implementing ordered, reliable multicast out of simple point-to-point primitives is non-trivial (whether based on synchronous or asynchronous point-to-point primitives), and deriving such protocols has been an active research topic for the last decade. Moreover, many networks, including Ethernet, support *hardware multicast* which transmits a multicast as quickly as a single point-to-point message. Once a multicast has been translated into numerous point-to-point messages, it is extremely difficult—or at least very expensive—to reconstitute the multicast destination sets. Therefore, distributed languages that wish to support multicast applications should provide a multicast primitive. Although one could imagine a system that provided a synchronous multicast—one in which a sender synchronized with all receivers—such a primitive would be extraordinarily expensive, especially if it could be used in selective communication [Knabe 1991]. DML provides an asynchronous multicast.

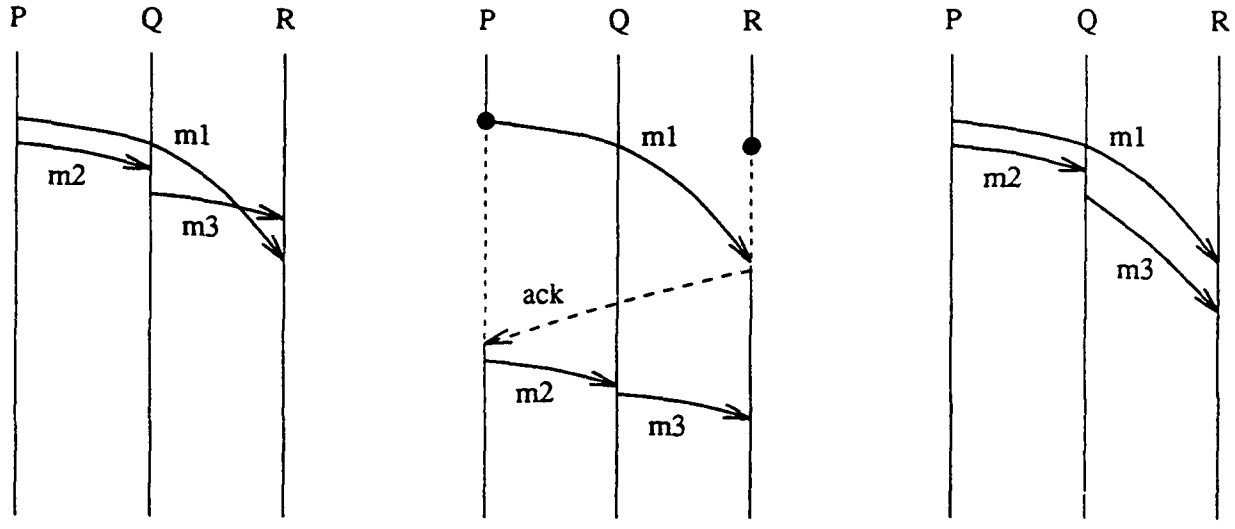
IV. Multicast Ordering

An important property of synchronous communication is that any operations executed by the sender upon return from the synchronous send are known to occur after the receipt of the message at the destination. That is, synchronous send operations force particular orderings on the events that follow a communication at both sender and receiver. Often these orderings are too strict and can be substituted with cheaper orderings. DML port groups support three kinds of ordering: FIFO, causal, and total. Causal ordering is the least widely understood and the most interesting from theoretical and performance aspects.

A Causal Ordering

Causal ordering is based on the potential causality (or "happens before") relation [Lamport 1978]. The *causal ordering* method ensures that if the sending of m' causally follows the sending of

m, then each destination that receives both messages delivers m before m' [Birman *et al.* 1991b]. Consider the example in Figure 3 concerning three threads, P, Q and R. P sends message m_1 to R and then sends m_2 to R. Upon receiving m_2, Q sends m_3 to R. It is probably intended that R receive m_1 before m_3; at first glance, it seems that synchronous message passing would be necessary to enforce this behavior. However, it is much more efficient to use causally ordered asynchronous messages. Causal ordering forces R to receive m_1 before m_3 without requiring an explicit acknowledgement message. Causal ordering can be used in many instances where a synchronous protocol might at first appear necessary.



Unordered communication

Synchronous communication

Causal asynchronous communication

Fig. 3. Causal ordering

Using asynchronous send operations, we can build a synchronous abstraction from a pair of asynchronous message operations and use it when synchronous properties are desired. In DML, this abstraction would have the same status as the built-in asynchronous operations. Of course, one could argue that asynchronous operations can be implemented using synchronous operations and extra threads. But this implementation incurs the extra message transmission costs we wish to avoid in the asynchronous case [Birman and van Renesse 1992].

```
signature ORDERINGS =
  sig
    type causal_dom
    type total_dom
    datatype ordering = FIFO
                      | CAUSAL of causal_dom
                      | TOTAL of total_dom

    val causal0: causal_dom
    val mkTotDom: causal_dom -> total_dom
    ...
  end;
```

Fig. 4. Ordering signature

B *Causal Completeness*

DML's causal ordering implies a *causal completeness* property. Informally, this property guarantees that there are no gaps in the middle of a process's causal message history due to crashes, although the history may be truncated by a crash. More precisely, causal completeness states that for any message, m, delivered to process P:

- all m's causal predecessors that are destined for P will have been delivered to P, and
- all m's causal predecessors not destined for P will *eventually* be delivered to their surviving destinations, despite subsequent node crashes.

There are a range of implementation techniques that satisfy this property.

C *Ordering Declarations*

In DML, orderings are associated with port groups. Recall that the `mkGrp` function from Figure 2 had an `ordering` argument. The signature for orderings is in Figure 4.

Elements (domains) of type `causal_dom` and `total_dom` may be used to enforce ordering schema among multiple port groups. For instance, if two port groups were declared to be `CAUSAL` with respect to the same `causal_dom`, data multicast through either the group is ordered relative to data multicast in both groups.

V. Message Stability

One important property of synchronous communication is that when a synchronous send operation returns, its message has been reliably received by the destination node(s). In particular, the

subsequent failure of the sender cannot prevent delivery of the message to the receiver. Message stability can be expensive, because a sender must wait after transmitting a message until an acknowledgement is received from a receiver. As computer networks and I/O hardware become faster, they provide larger message throughput but spend relatively more time putting messages on the wire. A sender and receiver cannot use this higher bandwidth effectively if every communication must incur the round-trip packet latency of the network.

In practice, message stability need be achieved only at particular points in a stream of remote sends (*e.g.*, before dispensing cash from an automatic teller, before confirming a reservation to a ticketing agent, or before closing a file). DML's `portTransmit` operation does not guarantee message stability, but returns a CML `event` value, a token which can be redeemed for a guarantee that the message has been received at its destinations.

In many other cases the multicast atomicity property suffices. The `portTransmit` operation implements multicast atomicity when sending to multiple destinations.

VI. Ordered Events

One of the more remarkable features of Concurrent ML is its support of *first-class synchronous operations* [Reppy 1991]. Many concurrent operations (*e.g.*, timing out, message transfer, detection of thread termination) block, and blocking operations are not easily embedded in new synchronous operations. In CML, concurrent operations return *event* values[1] instead of blocking, which permits new synchronous constructs to be constructed easily and elegantly.

Unfortunately, CML events are unordered and therefore not appropriate for some DML operations. For instance, consider the CML function

```
select: 'a event list -> 'a
```

which synchronizes upon exactly one element of its parameter list. If more than one element is ready for synchronization, the function chooses one of them.

Consider using CML's `select` on a list of two receive events ordered by the causal ordering property. If both receive events were available when the `select` was executed, it would be inappropriate to choose nondeterministically between them, and instead, the prior event should be selected. One could pass all ordered receives through a thread that released them one at a time, but doing so imposes a total order when only a partial order suffices, thereby restricting concurrency and, perhaps, leading to deadlock. Indeed, the Isis system [Birman *et al.* 1991a], from which these ordering and multicast properties are derived, constrains each node to receive a totally ordered message stream.

[1] An event should be thought of as the *potential* to block. In particular, it is not a *future*, and functions which return event values need not initiate a blocking operation.

```
signature ORD_EVENT =
  sig
    structure CML: CONCUR_ML
    type 'a ord_event

    val resolve: 'a ord_event list -> 'a ord_event
    val unord: 'a ord_event -> 'a CML.event

    val sync: 'a ord_event -> 'a
    val wrap: 'a ord_event * ('a -> 'b) -> 'b ord_event
    ...
  end
```

Fig. 5. Ordered event signature

In contrast, DML provides *ordered events* that, while similar to CML events, behave as elements of a partial order. Part of the ORD_EVENT signature is in Figure 5. The `resolve` operator selects between ordered events in a manner consistent with the underlying partial order. The `unord` function strips ordered events of their ordering information, and the `sync` and `wrap` operators operate like their CML counterparts. Ordered events allow DML to inherit the nice features of CML while preserving the information required to support ordered asynchrony in a pre-emptive environment.

VII. Example: Replicated processing

A more substantial example will illustrate both the ordering and stability issues. Suppose we are given a distributed program that communicates using only point-to-point channels (see Figure 6); consider how we can make it tolerate single crash failures by replicating each node [Alsberg and Day 1976]. A sketch of a non-fault-tolerant implementation of the `POINT_TO_POINT` signature appears in Figure 7.

In the fault tolerant version we replace each node P by a pair (P, P') consisting of a primary node and a backup which will take over the role of the primary should it fail. The backup maintains its internal state equivalent to the primary's, and, upon the primary's failure, produces a sequence of output messages consistent with the state of the primary at the instant it crashed. To achieve this, we arrange for the backup to observe exactly the same input messages as the primary, in the same order. Where the execution of the primary is nondeterministic (*e.g.*, because of pre-emptively scheduled threads), we must ensure the backup takes the same nondeterministic decisions. We will concentrate on the ordering and atomicity properties of internode communication, ignoring other

```
signature POINT_TO_POINT =
sig
  signature PortGroup: PORT_GROUP

  type 'a remote_channel
    (* Only one dest_port is permitted per channel. *)

  val mkChannel: unit -> 'a remote_channel
  val mkSrc: 'a remote_channel -> 'a PortGroup.src_port
  val mkDest: 'a remote_channel -> 'a PortGroup.dest_port
  val portSend: 'a PortGroup.src_port * 'a -> unit
  val portAccept: 'a PortGroup.dest_port -> 'a
end
```

Fig. 6. Signature for simple point-to-point remote communication

```
functor SimpleP2P(structure PortGroup: PORT_GROUP): POINT_TO_POINT =
  struct
    structure PortGroup = Portgroup

    local open PortGroup PortGroup.Ports
    in
      type 'a remote_channel = 'a port_group,

      fun mkChannel () = mkGrp Ord.FIFO;
      fun mkSrc g = #1 (PortGroup.mkSrc (g, ()));
      fun mkDest g = #1 (PortGroup.mkDest (g, ()));
        (* In practice there would be code to ensure only one
           dest_port per channel. *)
      fun portSend (src, data) = (CML.sync (portTransmit (src, data)); ());
      fun portAccept dest = OrdEvt.sync (portReceive dest);
    end;
  end
```

Fig. 7. Non-fault-tolerant implementation of point-to-point channels

```
signature PROCESS_PAIR =
  sig
    val primary: bool ref
    type schedule (* Information about nondeterministic decisions made
                     by primary *)
    val backup_chan: schedule port_group
  end

functor ResilientP2P(structure PortGroup: PORT_GROUP
                     and ProcessPair: PROCESS_PAIR): POINT_TO_POINT =
  struct
    structure PortGroup = Portgroup

    local open PortGroup PortGroup.Ports
    in
      type 'a remote_channel = 'a port_group,

      fun mkChannel () = mkGrp Ord.TOTAL;
      fun mkSrc g = #1 (PortGroup.mkSrc (g, ()));
      fun mkDest g = #1 (PortGroup.mkDest (g, ()));
      (* In practice there would be code to ensure only one dest_port
         at each of the primary and backup. *)

      fun portSend (src, data) =
        if ProcessPair.primary then
          (CML.sync (portTransmit (src, data)); ())
        else ();

      fun portAccept (src) =
        OrdEvent.sync (portReceive src);
    end;
  end
```

Fig. 8. Fault tolerant implementation of point-to-point channels.

issues including how the backup would use a `meta_dest_port` to notice the failure of the primary and how to handle nondeterminism due to thread scheduling and external I/O [Borg *et al.* 1989, Birman *et al.* 1991a].

We represent each point-to-point communication channel in the original program by a port group containing two `dest_port`'s, one owned by each of the primary and backup nodes (see Figure 8). The backup reads these messages and performs the same actions as the primary, except that output messages from the backup are suppressed. When the primary fails, the backup assumes the primary's role, setting the variable `ProcessPair.primary` to `true`. In addition there is a point-to-point port group (`ProcessPair.backup_chan`) from the primary to the backup. The primary uses this connection to send messages to the backup that resolve any nondeterminism, for

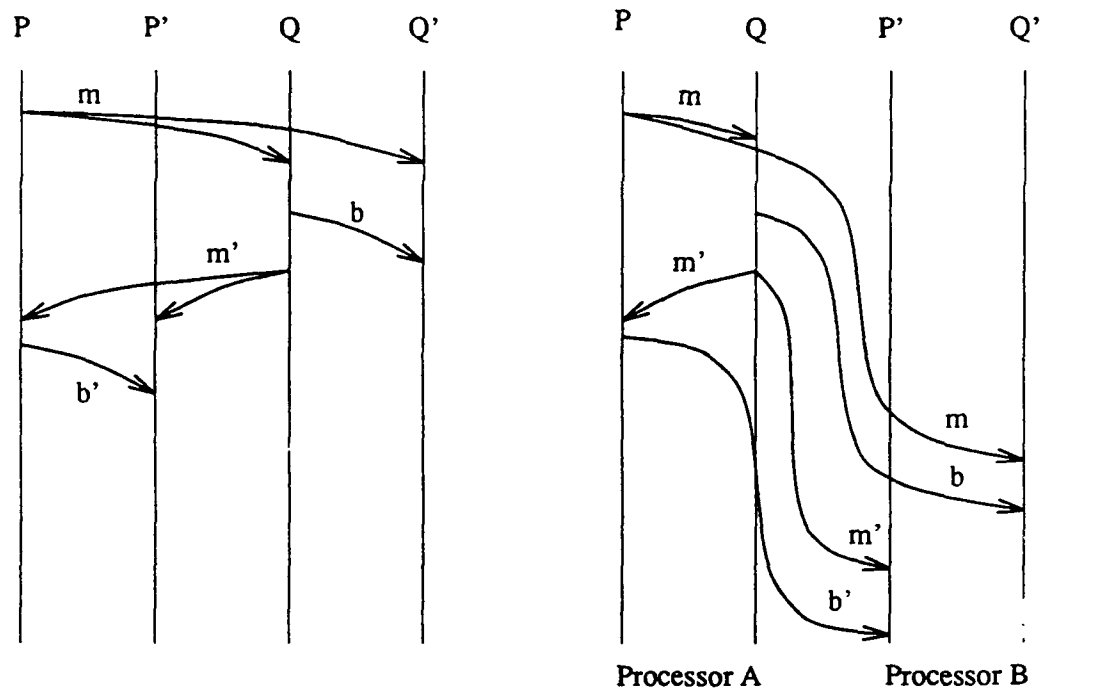


Fig. 9. Message patterns in fault-tolerant process pairs.

instance, caused by internal scheduling decisions made in the primary.

A point-to-point message, m, from node P to Q in the original program is transformed into a multicast from P to Q and its backup Q' (see Figure 9). Using a totally ordered multicast, Q and Q' will receive this message in the same order, relative to other multicasts.

We must also ensure that the scheduling message, b, sent by Q over the backup channel is delivered at Q' after the causally preceding multicast, m. If synchronous, non-causal communication is used, P must wait when it transmits m until both Q and Q' have received and acknowledged the message. If causally ordered communication is used throughout, P can initiate the multicast and immediately continue processing while the message is still being delivered. The causal ordering protocol includes information in message b identifying m as a causally preceeding message that must be delivered first. With synchronous, non-causal communication, an extra message delay is inserted into the critical path of the program.

This performance penalty is more severe than at first appears. To increase the independence of failures we might locate the primary and backup of a pair in physically separate locations. To improve performance in the normal case (no failures) we might locate all the primaries near each other (perhaps on the same computer). Now the synchronous approach performs much more slowly than the asynchronous causal approach. In the synchronous case, all communication will

occur at the speed dictated by the primary-backup connections. In the asynchronous approach, communication among the primaries can proceed at close to the speed attainable in the original non-fault-tolerant program. Communication between primaries and backups can proceed slower, in the background. Because messages are asynchronous, multiple primary-backup messages can be combined automatically into a smaller number of large network packets which will make much more efficient use of the network. We are not limited by network latency (round trip packet times), but bandwidth. Bandwidth scales much better than latency on almost all network technologies.

But what happens if a primary fails before its backup node has received all of the messages destined for it? The causal completeness property ensures that for any message received by another node, the causally preceding messages destined for the backup will be delivered to it. Thus, if any "evidence" of an action taken by the primary has been observed by another node in the program, the backup will receive any prior messages from the primary. Conversely, if no evidence of the final few actions of the failed primary is visible, then we can present the illusion that those actions never took place. There remains the possibility that the backup will send out some messages that duplicate the last few messages sent by the primary. These duplicates can be detected easily at the destinations using sequence numbers.

We see that asynchronous communication would not be feasable in this example without the causal ordering and completeness properties, which ensure that asynchronous messages are delivered in the correct order even when they are delayed and nodes crash, and the multicast atomicity property, which ensures that either all destinations receive the multicast, or none of them do.

VIII. Conclusion

We have argued that asynchronous communication, coupled with appropriate ordering and failure properties, permits higher network performance than synchronous communication. As networks become faster relative to processors, asynchronous communication will be essential. Additionally, we have argued for multicast as a communication primitive in support of fault tolerant and parallel programming. DML's ordering properties, such as total and causal orderings, are essential to both multicasts and asynchronous communication. They reduce the number of possible event histories, without compromising performance.

REFERENCES

[Alsberg and Day 1976]

P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, pp. 627–644, October 1976.

[Appel and MacQueen 1991]

A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pp. 1–26. Springer-Verlag, August 1991.

[Auerbach *et al.* 1991]

J. S. Auerbach, D. F. Bacon, A. P. Goldberg, G. S. Goldszmidt, M. T. Kennedy, A. R. Lowry, J. R. Russell, W. Silverman, R. E. Strom, D. M. Yellin, and S. A. Yemini. High-Level Lanugage Support for Programming Reliable Distributed Systems. Technical Report RC 16441, IBM T. J. Watson Research Center, January, 1991.

[Birman and van Renesse 1992]

Kenneth P. Birman and Robbert van Renesse. RPC Considered Inadequate, 1992. In preparation.

[Birman *et al.* 1991a]

Kenneth P. Birman, Robert Cooper, and Barry Gleeson. Design Alternatives for Process Group Membership and Multicast. Technical Report 91-1257, Department of Computer Science, Cornell University, December 1991.

[Birman *et al.* 1991b]

Kenneth P. Birman, Andre Schiper, and Patrick Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, **9**, 3, pp. 272–314, August 1991.

[Borg *et al.* 1989]

A. Borg, W. Blau, W. Gretsch, F. Herrmann, and W. Oberle. Fault Tolerance under Unix. *ACM Transactions on Computer Systems*, **7**, 1, pp. 1–23, February 1989.

[Knabe 1991]

Frederick Knabe. A Distributed Protocol for Channel-Based Communication with Choice, September 1991. In preparation.

[Krumvieda 1991]

Clifford D. Krumvieda. DML: Packaging High-Level Distributed Abstractions in SML. In Robert Harper, editor, *Proceedings of the Third International Workshop on Standard ML*, Department of Computer Science, Carnegie Mellon University, Sept. 26–27 1991.

[Krumvieda 1992]

Clifford D. Krumvieda. Expressing Fault-Tolerant and Consistency-Preserving Programs in Distributed ML. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[Lamport 1978]

Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, **21**, 7, pp. 558–565, July 1978.

[Matthews 1991]

David C. J. Matthews. A Distributed Concurrent Implemenation of Standard ML. Technical Report ECS-LFCS-91-174, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, August 1991.

[Milner *et al.* 1989]

Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1989.

[Reppy 1989]

John H. Reppy. First-class Synchronous Operations in Standard ML. Technical Report 89-1068, Department of Computer Science, Cornell University, December 1989.

[Reppy 1991]

John H. Reppy. CML: A Higher-order Concurrent Language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 293–305, June 1991.

Summaries of Poster Session Presentations

A Verification Environment for ML Programs

A. Cant and M. A. Ozols
Information Technology Division
Defence Science and Technology Organisation
PO Box 1500
SALISBURY 5108
SOUTH AUSTRALIA
Email: cant,maris@itd.dsto.oz.au

Abstract

A verification environment for Standard ML programs is described. The system is constructed within the Isabelle theorem prover, directly from the operational semantics definition of $\mathcal{F}$, the functional subset of the Core of SML. Both the static and the dynamic semantics of $\mathcal{F}$ are incorporated. Simple proof procedures can prove $\mathcal{F}$ programs correct by inferring the result of evaluation or elaboration demanded by the Definition. The work will benefit those trying to understand the Definition of SML, and experts who wish to explore possible modifications and extensions.

1 Introduction

In this paper we report on progress towards the construction of a verification environment for reasoning about ML programs. The system we describe captures the formal operational semantics definition of Standard ML [1] within the theorem prover Isabelle. Our work has arisen from the desire to explore the role of formal semantics in the verification process, especially in the context of critical software, in which a single error could have disastrous consequences. It complements work described in [2], where we explored the role of denotational semantics in the verification process.

ML has evolved from its early days as a special purpose language for theorem provers [3, 4] to the point where it is a serious programming language, expressive enough for many real applications, and with numerous desirable features (such as strong typing, exception handling and modules) which modern software engineering and critical software development require. Although, to our knowledge, ML has not yet been used in critical software projects, we believe that the techniques described below are applicable to any language built on rigorous principles. They will also be applicable to the problem of automated reasoning about programs in process algebras such as CCS [5] and CSP [6].

The rest of this paper is structured as follows. In Section 2 we describe the structure of the Definition of SML. In Section 3 we give a brief overview of Isabelle. Section 4 describes how our system is constructed. Finally, Section 5 discusses the results and give suggestions for further work.

2 The Definition of SML

The rigorous basis for Standard ML is formally given by the Definition [1], and its companion Commentary [7]. To quote from [1]:

> "This document is a formal description of both the *grammar* and the *meaning* of a language which is both designed for large projects and widely used. ... At a time when it is increasingly understood that programs must withstand rigorous analysis, particularly for systems where safety is critical, a rigorous language presentation is important even for negotiators and contractors, for a robust program written in an insecure language is like a house built upon sand."

The Definition uses operational semantics [8] to describe the meaning of phrases in the language. This method of language definition is a useful guide for the implementer of an interpreter or compiler for the language. It is especially useful for describing the semantics of concurrent languages, where the denotational semantics may be technically difficult and less easy to mechanise.

The Definition first presents the syntax for SML (both for the Core language and the Modules System), introducing various identifier and phrase classes. The static semantics (elaboration) is then given in detail. This involves a rich set of simple and compound semantic objects (such as environments, types etc). Elaboration of a phrase is expressed by a sequent of the form

$$E \vdash \text{phrase} \Rightarrow \text{result}$$

where typically E is an environment, and the result may be a type or an environment. The 102 inference rules capture all the possible inferences among these sequents.

The dynamic semantics is then given a similar, but quite separate, treatment. The fact that evaluation and elaboration can be dealt with independently is an important aspect of SML. Often the same terminology gets used in both the static and dynamic semantics but has a different meaning in each case, such as "variable environment". Static and dynamic semantics meet at the level of programs, where the evaluation of a program is only carried out if it elaborates successfully.

Standard ML has a number of phrase classes which are derived forms. For example, the program phrase **case exp of match** is defined to be the more primitive language expression **(fn match) (exp)**. Other examples of derived forms are **if**, **andalso** and **orelse**, as well as lists and tuples. Inference rules only need to be given for the phrases in the so-called *bare* language.

Because of their formal nature, and the size of the language, the Definition and Commentary are not light reading, and are aimed at implementers and ML experts more than the general reader. The Definition allows one in principle to explore language semantics, but detailed proofs done on paper using all the 196 inference rules are far too laborious: we believe that machine support is essential to be able to do this.

In constructing a system for reasoning about SML programs, our aim has been to capture the definition of SML in as natural a way as possible in a powerful proof assistant. One might consider using Prolog for this purpose [9]. However, we have chosen to use the theorem prover Isabelle, which has a number of advantages over Prolog: it allows the use of concrete syntax; theories can easily be related to familiar logics (such as First Order Logic and Higher Order Logic); and one has much finer control of proofs steps and search. We believe, therefore, that Isabelle is an ideal tool for reasoning about operational semantics.

3 Isabelle

The theorem prover Isabelle (itself constructed using SML!) has been under development by Larry Paulson at the University of Cambridge since 1986 [10, 11, 12]. Isabelle is a generic theorem prover, with an expressive *meta-logic*, in which the inference rules and axioms of *object logics* can be formulated. Isabelle emphasises the natural style of reasoning, and thrives on inference rules such as those of operational semantics. It is this fact which has been exploited in our work.

Isabelle carries out goal-directed proofs. A *proof state* consists of a *goal*, along with a number of subgoals whose validity establishes that of the goal. Proving a goal involves reaching a proof state with no subgoals, by means of the application of *tactics*, which transform proof states to new proof states. In Isabelle a tactic may fail, or return one or more (and possibly even an infinite number) of new proof states.

The most important tactic is *resolution*: `resolve_tac thms i` tries each theorem (object logic rule) in the list `thms` against subgoal i of the proof state. For a given rule, say

$$[\![B_1, \ldots, B_k]\!] \Longrightarrow B$$

resolution can form the next state by unifying the conclusion with the subgoal, replacing it by the instantiated premises. (Note that unification in Isabelle is full higher-order unification [13]). Thus if the subgoal is

$$[\![A_1, \ldots, A_n]\!] \Longrightarrow A$$

and A can unify with B, resolution will produce the following new subgoals:

$$[\![\overline{A_1}, \ldots, \overline{A_n}]\!] \Longrightarrow \overline{B_1}$$

$$\ldots$$

$$[\![\overline{A_1}, \ldots, \overline{A_n}]\!] \Longrightarrow \overline{B_k}$$

in which the overbars denote the resulting formulae after instantiations have been made.

Isabelle also has a number of *tacticals*, used for building new tactics from basic tactics, for example:

tac1 THEN tac2 (sequencing)
tac1 ORELSE tac2 (choice)
REPEAT tac (iteration)
DEPTHFIRST pred tac (search)

The tactic `DEPTH_FIRST pred tac` performs a depth-first search for a proof-state satisfying `pred`. Usually `pred` is taken to be "no subgoals", so that the tactic will search for a proof of the original goal.

Isabelle also has answer extraction available, via so-called *scheme variables*. These variables can be part of a goal; as tactics are applied the scheme variables may be instantiated during the proof.

4 Description of the System

At present, we have built a system for a non-trivial subset $\mathcal{F}$ of Standard ML — essentially the pure functional (side-effect free) subset of the Core Language, including pattern matching, functions as first-class objects and recursion. Therefore, we have excluded imperative features such as reference variables, assignments, and exceptions, as well as the modules system. However, what remains is still an extremely rich language.

The syntax, semantic objects and inference rules of $\mathcal{F}$ are constructed as a new theory which is a (typed) Isabelle object logic $\mathcal{L}$[1]. The types of the logic are expressions, declarations as well as objects such as values, environments, identifiers etc.

4.1 Syntax

Isabelle allows the user considerable freedom in the choice of concrete syntax. It is a great aid to understanding to be able to use a nice concrete syntax which is also strictly maintained during a proof. Our aim has been to use wherever possible the precise syntax of SML For example, functions are given in SML by:

exp ::= ...
fn match
match ::= pat => exp ⟨ | match⟩

which is easily expressed as the following Isabelle syntax declaration:

```
Delimfix ("fn _", Match --> Exp, ...),
Delimfix ("_ => _", [Pat, Exp] ---> Match, ... ),
Delimfix ("_ => _ | _",
          [Pat, Exp,  Match] ---> Match, ...),
```

Note that **Match** and **Exp** appear as new types in the logic; thus Isabelle's type-checking will catch syntax errors.

However, variables (and also value constructors and special constants) do need to be explicitly marked, for example: `fn {var x, var y, ...} => var x + var y`. Note that we are at liberty to overload notation: for example, we can use curly brackets for record expressions, record patterns and record values, just as the Definition requires. The same holds for lists and pairs. Isabelle can tell what is meant from the context.

Another of Isabelle's strengths is the ability to include derived forms by means of parse translations (which take a simple concrete syntax phrase such as **case exp of match** into the internal representation of the appropriate bare language phrase), and print translations (which reverse the process). Once these translations have been provided, we can use the derived forms freely in goals, and they will be correctly dealt with.

[1] Actually, $\mathcal{L}$ is an extension of First Order Logic.

4.2 Semantic Objects

In the spirit of [1], elaboration (static semantics) and evaluation (dynamic semantics) are treated separately; this is reflected in the design of the system. Thus, separate Isabelle theories are maintained: one for elaboration and one for evaluation. They have in common the syntax of $\mathcal{F}$ itself.

The various semantic objects, such as values, environments etc are themselves given an appropriate concrete syntax, and their properties described by means of inference rules within the logic $\mathcal{L}$ for the various operations defined on these domains.. For example, variable environments are of the form

$$\{|\ (x_1, v_1), ..., (x_n, v_n)\ |\}$$

for evaluation, and of the form

$$\{|\ (x_1 : t_1), ..., (x_n : t_n)\ |\}$$

for elaboration.

Typical inference rules are the following, which describe the operation of looking up a value in a variable environment:

```
val lookup =
    [("lookup1_rule",
        " lookup (x, {| (x,v) |}, v)"),
     ("lookup2_rule",
        " lookup (x, {| (x,v), bindseq |}, v)"),
     ("lookup3_rule",
        " lookup (x, {| bindseq |}, v) ==>
          lookup (x, {| (y,w), bindseq |}, v)") ];
```

4.3 Static Semantics

For the most part, the inference rules capturing the static semantics of $\mathcal{F}$ are easily expressed in Isabelle. For example, here is the Rule 6 for the elaboration of the atomic expression **let dec in exp end**. This is expressed as the following Isabelle fragment:

```
("Let_ty_rule",      (* Rule 6 *)
    "[| C |- dec -> E ; combine (C,E,C'); C' |- exp ->  t |]
          ==>  C |-  let dec in exp end ->  t ")
```

where `combine` is the operation which combines contexts together. Note the way that sequents are written in these rules.

In the static semantics, the trickiest aspect to model is polymorphism. We need to set up a number of inference rules to implement the considerable machinery of type schemes, type instances and closures in order allow sound polymorphic typing.

4.4 Dynamic Semantics

Once again, it is straightforward to express inference rules. Consider, for example, the Rule 108, for the evaluation of the above expression:

```
("Let_rule",      (* Rule 108  *)
    "[| E |- dec -> E'; combine (E,E',E''); E'' |- exp -> v |]
       ==> E |- let dec in exp end -> v ")
```

The dynamic semantics needs to have function closures, in order to have correct call-time environments. Recursion demands the unfolding operation on environments. These have been incorporated in the system.

4.5 Proof Procedures

The proof procedures for reasoning about programs are quite simple Isabelle tactics which capture the way one would construct an inference tree [7]. Starting from the initial goal (the root of the tree), we

keep resolving with the appropriate inference rules, simplifying environments as we go, until all language phrases have disappeared. We then simplify using the inference rules for semantic objects until we reach the leaves of the tree. Isabelle keeps track of the instantiations made as we go. Thus we can prove $\mathcal{F}$ programs correct by inferring the result of evaluation or elaboration demanded by the Definition. Proofs of ML programs involving pattern matching may require backtracking search. To accommodate this, our proof procedures use the depth-first search tactical.

Polymorphic type inference, as one would expect, requires special handling. However, the actual work of inferring the most general type of an expression is aided by Isabelle's scheme variables, which can then be instantiated to type variables.

5 Conclusions and Suggestions for Future Work

The system is easy to modify, and is quite efficient, because it exploits Isabelle's liking for inference rules, and keeps costly rewriting to a minimum.

We believe that the present work will benefit a number of people. It should be of help to beginners trying to understand the Definition of SML, as well as implementers. It can assist experts who wish to explore possible design changes and extensions to the language, by aiding reasoning about how the various phrases will interact. We also believe that the system will benefit those interested in program verification, equivalences and transformations.

The user interface is still under development. Ideally, it should offer the user a range of choices. A minimal interface would be to mimic that of an ML interpreter, presenting only the value and type of the most recently declared variable(s). The maximal interface would be to describe fully the proof of each sequent, showing also the context before and after the evaluation or elaboration. This information can quickly become too much to cope with.

In future work, we would like to extend the system to allow for a larger subset of SML. Exceptions and imperative features will require proper handling of the state and exception conventions [1, 7], while the inclusion of the modules system would allow experiments with subtle aspects of signature matching and elaboration of functors and signature expressions.

Further work needs to be done on general proofs of correctness of algorithms written in SML, for which we need to have an expressive specification language. Since our system is built on Isabelle's First Order Logic, we already have the basis of such a language, while our system could easily be built on Isabelle's Higher Order Logic, if required. Such specification constructs are under investigation as part of the Extended ML language of Sannella and Tarlecki [14], and in current work of Gene Rollins, Jeannette Wing, and Amy Moormann Zaremski at CMU on Larch/ML.

Another important line of investigation is that of reasoning about program equivalences and transformations [15, 16]. The operational semantics rules which define SML will need to be strengthened to allow such reasoning.

We are also planning to apply our methods to concurrency, where operational semantics is frequently used to give the meaning of language constructs: possibilities include CML [17], CCS [5] and the tasking model of Ada.

6 Acknowledgment

The authors wish to thank Malcolm Newey (Australian National University) and Larry Paulson (University of Cambridge) for their helpful suggestions.

Bibliography

[1] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[2] A. Cant and M. A. Ozols. The Role of Denotational Semantics in Program Verification. *Formal Aspects of Computing (to be submitted)*, 1992.

[3] R. Milner M. Gordon and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Lecture Notes in Computer Science, No 78. Springer-Verlag, 1979.

[4] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[5] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentuce-Hall.

[7] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[8] G. D. Plotkin. A Structural Approach to Operational Semantics. Report, University of Aarhus, Denmark.

[9] R. Sethi. *Programming Languages, Concepts and Constructs*. Addison-Wesley, 1989.

[10] L. C. Paulson and T. Nipkow. *Isabelle Tutorial and User's Manual*. Computer Laboratory, University of Cambridge, June 1990.

[11] L. C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[12] L. C. Paulson. Isabelle: The Next 700 Theorem Provers. *Logic and Computer Science (P Odifreddi, ed)*, pages 361–385, 1990.

[13] G. Huet. A Unification Algorithm for Typed λ–calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[14] D. T. Sannella and A. Tarlecki. Towards Formal Development of ML Programs: Foundations and Methodology. Report EFS-LFCS-89–71, University of Edinburgh, 1989.

[15] R. J. R. Back and J. von Wright. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing Systems*, 20:799, 1990.

[16] R. Roxas and M. C. Newey. Proof of Program Transformations. HOL '91 User Meeting, Aarhus, Denmark, Australian National University, 1991.

[17] J. H. Reppy. *Concurrent Programming with Events: The Concurrent ML Manual*. Dept of Computer Science, Cornell University, Ithaca NY, October 1991.

Expressing Fault-Tolerant and Consistency-Preserving Programs in Distributed ML

Clifford D. Krumvieda[†]
cliff@cs.cornell.edu

Computer Science Department, Cornell University, Ithaca, NY 14853

ABSTRACT

Many of the programming problems particular to distributed environments involve violations of *consistency*. For example, some applications require that replicated data copies be kept identical. If explicit failure information is provided by the programming system, failure notifications must be ordered so that distributed components maintain a consistent view of system membership. We are currently adding features to the Standard ML (SML) programming language that help distributed programmers preserve consistency. Our system, called Distributed ML (DML), introduces *ports* and first-class *port groups* as a method of abstracting asynchronous multicasts. DML provides *meta ports*, multicast orderings, and ordered events to help programmers reason about the dynamic distributed environment. This paper contains three examples that illustrate DML's syntax and operational semantics.

I. CONSISTENCY

There are at least two reasons that programmers write distributed programs. First, many applications require parallelism. For instance, distributed databases and electronic mailers are inherently distributed; other applications need real concurrency to execute quickly. Second, some applications must behave correctly despite component failures. Running a single program on multiple, independently-failing computers is an appealing way to achieve resiliency. File servers and air traffic controlling programs are examples of applications requiring high availability.

Unfortunately, naive attempts to implement fault-tolerance and concurrency with distributed systems can lead to violations of *consistency* [Birman 1991]. There are numerous consistency properties, and they vary among applications. For instance, some applications maintain replicated data sets which must remain consistent with each other. Data consistency is threatened if replicas receive concurrent updates in different orders. Also, many applications divide work among active processes and therefore require a consistent view of outstanding tasks and available processors. Component failures may lead to inconsistent system "views" of active processes.

In a companion paper [Cooper and Krumvieda 1992], we discussed the Distributed ML (DML) programming language and justified its asynchronous multicast primitives. Port groups, multicast orderings, and ordered events are several of the tools that DML programmers can use to maintain consistency in a distributed environment. In this paper, we present three DML programming examples and argue that explicit failure information through *meta ports* can be used to maintain consistent views of system membership.

II. EXAMPLE 1: RPC MULTICAST

The *client/server model* is a common paradigm in fault-tolerant distributed programming. In this model, a group of *server* programs control a resource (e.g., a database or a speedy processor) while *client* programs

[†]The author is supported by a National Defense Science and Engineering Graduate Fellowship sponsored by the Air Force Office of Scientific Research/AFSC, United States Air Force, under Contract F49620-86-C-0127.

communicate with the servers to access the resource. Often, the servers are replicated to provide fault tolerance. Here, communication is *two-way*; that is, clients *initiate* communications with a server and servers send *reply* messages to waiting clients. If a server group has only one member, this communication style is called RPC, or *Remote Procedure Call*. If there are several servers, the style is *RPC Multicast*; often, distributed toolkits (such as Isis [Birman and Joseph 1987]) directly support RPC Multicast.

One possible DML implementation of RPC Multicast would require a single port group shared by both clients and servers; clients would ignore messages sent by other clients. However, this solution requires each client to have a separate `dest_port` and would be too inefficient when clients greatly outnumber servers (which is typical).

Instead, we define a type `('a, 'b) rpc_group`. Clients send requests of type `'a` to servers, while servers send replies of type `'b` to clients.

```
type 'a rpc_reply = 'a * port_id
type ('a, 'b) rpc_request = 'a * (('b rpc_reply, unit) port_group)
type ('a, 'b) rpc_group = (('a, 'b) rpc_request, unit) port_group
```

Data sent through an `rpc_group` are `rpc_requests`, pairs which contain a request and a port group for replies. Replies are pairs containing reply data and a `port_id` identifying the replying server. All meta data in this example has type `unit`.

The function `request`, used by clients to initiate RPC multicasts, has type

```
val request: (('a, 'b) rpc_request, unit gview event) src_port * 'a ->
                                        'b rpc_reply dest_port * unit gview event
```

The first coordinate of the pair returned by `request` is a destination port used to queue server replies; each reply contains data of type `'b` and a `port_id` of the `dest_port` from which the request was received. The second half of the pair is a `unit gview event` which, when `CML.sync` is applied to it, yields the membership of the server group when the original message was delivered[1]. This information is enough to collect replies in a number of different ways, one of which we will examine in section IV.

The server uses the function `service` to reply to client queries.

```
val service: ('a, 'b) rpc_request dest_port * ('a -> 'b) -> unit
```

This function is used to register a "handler" function with a destination port. After a `service` call is evaluated, incoming requests are processed automatically; replies are calculated by feeding requests to the registered function.

The implementations of `request` and `service` are shown in Figure 1. At first glance, they seem to be extremely inefficient because the number of port groups required is linear in the number of requests (assuming a bounded number of servers). However, port groups with only one destination port can be implemented with point-to-point communication and are therefore cheap.

RPC Multicast clients receive a stream of reply messages and, to avoid blocking for messages that may never arrive, they must monitor the servers for failure.

III. Explicit Failure Information

DML is intended to be used in *asynchronous* distributed systems, that is, systems in which communication times are unbounded[2]. Unfortunately, many important problems that involve consistency among processes—

[1] The `unit` in `unit gview event` is the group's meta data type and will be discussed in section III.

[2] The word "asynchronous" is overloaded: *asynchronous* systems should not be confused with *asynchronous* communication. Asynchronous communication (in which data senders do not rendezvous with data receivers) can occur in synchronous systems and vice versa.

```
fun request (sp, msg) =
  let
    val rg = mkGrp FIFO;
  in
    (#1 (mkDest (rg, ())), portTransmit (sp, (msg, rg)))
  end;

fun service (dp, fcn) =
  (wrap (unord (portReceive dp),
         fn (msg,rg) => let val sp = #1 (mkSrc (rg, ()));
                        in portTransmit (sp, (fcn msg, portID dp)); ()
                        end);
   service (dp, fcn));
```

Fig. 1. Implementations of `request` and `service`

such as agreement on system membership and consensus on the value of a variable—cannot be solved in failure-prone asynchronous systems [Fischer *et al.* 1985].

The DML runtime system circumvents this impossibility result by providing a (scalable) system-wide *membership service* [Ricciardi and Birman 1991]. The membership service maintains the set of active DML nodes and can add and remove nodes dynamically. In particular, the service may delete nodes that appear to have failed (perhaps because they didn't acknowledge a series of messages), even if they may still be working properly.

DML port groups have *meta ports* that the membership service uses to inform programs of changes to the system membership. Consider a portion of the signature `PORT_GROUP` (more of which is defined in [Cooper and Krumvieda 1992]):

```
signature PORT_GROUP =
  sig
    structure Ports: PORTS

    ...
    type 'a gview
    type 'a monitor_event = SRC_CREATE of 'a * Ports.port_id
                          | SRC_FAIL of Ports.port_id
                          | DEST_CREATE of 'a * Ports.port_id
                          | DEST_FAIL of Ports.port_id

    val getView: ('a, 'b) port_group -> 'b gview
    val portMonitor: 'a gview -> ('a monitor_event * 'a gview) Ports.dest_port
    ...
  end;
```

A value of type `gview` contains port group membership information, and a `monitor_event` describes a specific event that triggered a change in the membership of some port group. All events provide the `port_id` of the created or failed port; create events also carry information provided when the port was created. The `getView` function uses the membership service, if necessary, to take a "snapshot" of its port group argument. The `portMonitor` function examines its `gview` argument, adds a meta `dest_port` to the appropriate group, and enqueues, at the beginning of the new `dest_port`, all changes (`monitor_events`) that the group underwent since the `gview` was constructed. Conceptually, the `dest_port` returned by the `portMonitor` functions is a stream of all membership changes—past, present, and future—to the port group described by its `gview` argument since the view was created.

```
fun procReply (n, (dp, ge)) =
  let
    fun state1 got =
      if length got = n then []
      else sync (choose [wrap (unord (portReceive dp),
                               fn (x,id) => x :: (state1 (id :: got))),
                         wrap (ge,
                               fn gv => state2 (destMonitor gv, got,
                                                numDests gv, 0))]);
    fun state2 (mdp, got, potential, noreply) =
      if length got = n orelse
         length got + noreply = potential then []
      else
         sync (choose [wrap (unord (portReceive dp),
                             fn (x, id) =>
                                x :: (state2 (mdp, id::got, potential, noreply)))
                       wrap (unord (portReceive mdp),
                             fn (DEST_FAILED id, _) =>
                                state2 (mdp, got, potential,
                                        if member id got then noreply
                                        else noreply + 1)
                                | _ => state2 (mdp, got, potential, noreply))]);
  in
     state1 []
  end;
```

Fig. 2. Implementation of `procReply`

IV. Example 2: Processing Replies

Recall the implementation of RPC Multicast of section II: clients are probably not interested in receiving a destination port and a `unit gview event` from their requests. Instead, they would prefer to use a library function to process incoming replies. We will define the function

```
val procReply : int * ('b rpc_reply dest_port * unit gview event) -> 'b list
```

`procReply` takes an integer n and the result of a `request` call and returns a list of reply values. This list will have length no greater than n, but may be less than n if fewer than n servers reply. Its implementation is in Figure 2.

A thread executing the `procReply` function is in one of two states, depending on whether or not the `unit gview` describing the query destination set has been built. Initially, the function is in `state1`; it waits for either a reply to be received on the reply group `dest_port` or for the `unit gview` build to complete. After the `gview` is completed, the function enters `state2` and monitors for replies or failed query group `dest_ports`. The variable `got` is a list of query `dest_port port_id`'s which have been acknowledged in replies. The integer `potential` is the maximum number of replies, while `noreply` indicates the number of failed `dest_port`'s that never replied.

V. Consistency and Membership

In sections II and IV we saw two examples of DML programs, and the latter motivated the need for explicit failure information. An important issue in systems that support consistency-preserving programs is the way they order failure information relative to other communication in the system. In DML, for instance:

1. When a `src_port` fails, its failure notification is ordered after all receive events for messages sent through the port.

2. If two failures occur simultaneously, their failure notifications are ordered identically at threads that receive them.

3. Any receive event of a port group will be ordered consistently with respect to a failure notification of any of the group's ports.

Properties two and three will be important in the next example.

VI. Example 3: Replicated State

Although the DML syntax does not provide for distributed references, they can be simulated using port groups. In this example, we describe an implementation of distributed references through *replicated state*; the value of the reference is replicated to provide fault tolerance and quick dereferences. However, updates are slow.

Define the type `'a dref_query` to be

```
datatype 'a dref_query = DEREF of ('a, unit) port_group
                       | UPDATE of 'a;
```

Copies of the replicated state communicate through `'a dref_groups`.

```
type 'a dref_group = ('a dref_query, 'a option) port_group;
fun mkDGrp () = mkGrp TOTAL;
```

Each distributed reference has an associated `dref_group`; threads dereference and update by sending messages through the appropriate group. For instance, the dereferencing function `!!` can be defined by

```
fun !! g =
  let
    val replyGr = mkGrp FIFO;
  in
    portTransmit (#1 (mkSrc (g, NONE)), DEREF replyGr);
    portReceive (#1 (mkDest (replyGr, ())))
  end;
```

Threads can install instances of the replicated state by calling the function `install`, defined in Figure 3. The implementation of `install` is notable for several reasons:

1. Dereference queries receive replies from *every* copy of the replicated state. A more intelligent (and lengthier) implementation would ensure that only one (local, if available) copy replies to each dereference.

2. `dref_groups` are totally ordered, so updates are received in the same order at each copy.

3. Updates and new `dest_port` creation events are ordered with respect to each other. This ensures that a new replica R processes the same stream of updates that old replicas process after observing R's creation.

4. There is no method for bootstrapping the example; in a complete implementation, there must be a function for creating the first replica.

```
fun install g =
  let
    val xg = mkGrp ();
    val (dest, view) = mkDest (g, SOME xg);
    val mdest = portMonitor view;
    fun server value =
      OrdEvt.sync (resolve [OrdEvt.wrap (portReceive dest,
                                         fn (DEREF g') =>
                                            (portTransmit (mkSrc (g', ()), value);
                                             server value)
                                          | (UPDATE x) => server x),
                            OrdEvt.wrap (portReceive mdest,
                                         fn (DEST_CREATE (SOME g')) =>
                                            (portTransmit (mkSrc (g', ()), value);
                                             server value)
                                          | _ => server value)]);
in
  server (OrdEvt.sync (portReceive (#1 (mkDest (xg, ())))))
end;
```

Fig. 3. Implementation of `install`

VII. Conclusions

We have illustrated that explicit failure information, coupled with the appropriate ordering properties, permits a flexible and convenient notation for high performance asynchronous communication.

References

[Birman 1991]

Kenneth P. Birman. Maintaining Consistency in Distributed Systems. Technical Report 91-1240, Department of Computer Science, Cornell University, December 1991.

[Birman and Joseph 1987]

Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, **5**, 1, pp. 47–76, February 1987.

[Cooper and Krumvieda 1992]

Robert Cooper and Clifford Krumvieda. Distributed Programming with Asynchronous Ordered Channels in Distributed ML. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[Fischer *et al.* 1985]

M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, **32**, 2, pp. 374–382, April 1985.

[Ricciardi and Birman 1991]

Aleta M. Ricciardi and Kenneth P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proc. 10th ACM Symp. on Princ. of Dist. Comp.*, pp. 341–353, Montreal, Quebec, Canada, Aug. 19–21 1991.

Implementing ML on the Fujitsu AP1000

Peter Bailey
Malcolm Newey
Department of Computer Science
Australian National University

Abstract

The CAP ML project seeks to develop a version of ML that is suitable for use on a distributed memory multiprocessor architecture such as the Fujitsu AP1000. Language extensions are proposed that have been developed in conjunction with a programmming methodology that is appropriate to that of a massively parallel computer whilst retaining a functional style. The implementation, which is based on the SML/NJ compiler and the SML2C compiler, is in progress. This paper focusses on the language design.

Introduction

When one contemplates implementing a functional language on a parallel computer, what first comes to mind is the well known advantage claimed for functional languages, that freedom from side-effects allows multiple processors to be utilised for the parallel execution of multiple arguments in any function call. There is also a long history of applicative language compilers which take advantage of the property of referential transparency, for a variety of purposes. For example, the implementation technique of graph reduction depends on it for achieving speedup through concurrency. However, experience indicates the opposite - it is *hard* to harness a multiprocessor for ML.

There are several lessons that have been learned that are important background as we embark on the enterprise of providing a functional programming capability on a machine such as the AP1000.

- Purely functional languages are not found suitable for large application programs. Lisp and ML are far more widely used than Miranda; although the latter has been widespread for some time, its niche in the market is in the education sector. Programmers still find important uses, in major applications, for global data structures that are updated, and for i/o operations that can't readily be characterised in functional terms.

- Although ML and Lisp both allow assignments to variables, the use of this feature is discouraged in situations where concurrency is expected. (ML discourages any such use by the very syntax of reference operations.) Thus the programmer should seek to structure a program so as to have as few routines as possible that side-effect the state.

- It is only sensible to spawn a process if it is likely to survive for a time which is long compared to its setup time. Experiments where a compiler does its best to recognise which sets of subexpressions can be executed concurrently show there is surprisingly little chance that substantial parallelism can be achieved in that way; some estimates suggest that typical Lisp programs would be unlikely to make effective use of just ten processors.

- The conclusion of all implementors of Lisp and ML compilers for parallel machines is that the programmer should design algorithms with concurrent execution in mind and give explicit advice about which parts of a program can be usefully mapped to processes. A common design decision is for the system to spawn processes *only* at places where the user advises that multiple arguments to a function should be evaluated concurrently.

- Most parallel Lisp and ML compilers have been implemented on shared memory multiprocessors and really depend on this fact for their success. The rule-of-thumb that has been suggested is that each process should run for some thousands of instructions to have a cost-effective existence. In many applications a programmer is likely to find many appropriate situations.

- In a distributed memory machine, each processor has its own memory and so the setup cost includes identifying (by following pointers) all relevant cells, copying these across a network, and initialising a new heap space with this data. The operation is something like a garbage collection so we would expect it to sometimes be most economical to simply copy all of the heap space from one processor to another. In the distributed memory case, spawning processes is likely to be appropriate only if such processes last for some millions of instructions, hardly a function call.

Fujitsu AP1000 Cellular Array Processor

The AP1000 is a highly-parallel *scalable* computer with distributed memory. Each cell consists of a SPARC CPU, a Weitek FPU, custom message controllers, and 16Mb of local memory. It has a front-end host Sun4/330 which connects to the cells. These in turn are connected by three separate high-speed networks - a 2D mesh-connection torus network, a broadcast network for one-to-n communications, and a synchronisation network. Typical sizes are 64, 128, 256, 512 and 1024 cells.

This new style of expensive machine is of the same class as the Thinking Machine Corporation's CM-5, Intel's iPSC and Touchstone Delta. They share the common features of difficulty of programming and paucity of software (beyond the C and Fortran compilers). The vanguard of applications are those based in the numerical analysis of scientific problems. In these applications, processors are typically allocated to independent calculations or computations on a hunk of an array. Typical techniques are Monte Carlo simulation and Finite Element methods, just as employed on SIMD machines.

Although MIMD machines exemplified by the AP1000 apparently have more scope for fast general purpose computation, we must learn how to use them less for Physics problems and more for AI (mathematics, knowledge, reasoning etc.).

Language Design

Fine grain concurrency, especially of the sort envisaged in graph reduction, is inappropriate where there is no shared memory. Because functions are closures, there will be potentially large amounts of heap space that must be copied from one processor to another, even for quite short expression evaluations. We see very coarse grain parallelism as necessary for the combination of distributed memory machine and applicative language.

It is expected that the programmer will carefully design algorithms with concurrency in mind and will take complete charge of the processes, both as syntactic objects and dynamically executing entities. Since we insist that the extensions to ML should retain the applicative flavour of the standard language as much as possible, processes will take arguments and yield a result.

The AP1000 style of architecture imposes considerable overheads on process creation, and thus we adopt a programming methodology that discourages the use of more processes than there are processors. The programmer should aim to create the required processes early in the execution of a program and expect that they will last for a time that is long compared to the process startup time.

In order that long-running processes can be used successfully, we must allow them to cooperate by passing information. In the case of the AP1000 this can only mean we provide communication between processes by message passing or by distributed shared memory; currently, we have chosen message-passing as the most efficient system to implement.

This design of the language is intended to support a two-level style of program structure where the top level is the initiation of 'actors' that interact with each other by message passing. Within these top level processes, the programming should resemble that of whole ML programs, where I/O is replaced by message traffic among the 'actors'. Use of messages is certainly not referentially transparent but the careful programmer can still structure each process in the applicative style and write most component functions to be side-effect free.

Based on these major design decisions, we present *paraML*, an extension of Standard ML that we claim is suitable for programming the highly parallel computers of the future. The AP1000 is a leading example, being a machine with many powerful processors, each with its own large local memory. The major extensions to Standard ML are given below.

Processes in ParaML

We make changes to two areas of ML to accomodate our notions of processes. We add a new sort of declaration (ie process definition) and two new forms of expression - one to create a process instance and one which gets the value computed by the process.

Process Definitions

The ML code that is the abstracted form of a process is declared in a way that is very similar to a function definition. The difference is that the external view of a process must reflect the fact that the process can receive messages on named channels. Thus the type of a process will have the form similar to $\alpha \rightarrow \beta \rightarrow \gamma$ where α is the type of the argument supplied, β is the type (a record type) of the n-tuple of channels, and γ is the type of the result of executing the process to completion. The complete syntax is given in Bailey [1] and a forthcoming manual, but the following is the usual way in which a single process form is defined.

```
declaration:
        define pdef (channel ch1, ch2, . . .  chn)  pat = exp;
binding:
        val pdef = prd:  'a->'b ->> 'c
```

`pdef` is bound to the process form described in the definition; it is an ML routine that takes an argument matching `pat`, that produces a result by evaluating `exp` and that receives messages

on the channels ch1, ch2, etc. pdef is called the process definition identifier. The words define and channel are new reserved words for ML and ->> is a new type constructor that is used in expressions for the types of process forms and process instances.

Of course, some processes that we wish to define will read no messages and so have no need of the channel list. However, this must be signified, like the unit argument to functions. The following syntax is appropriate in this case.

```
declaration:
        define pdef nochannels pat = exp;
```

The indicator, nochannels, is a new reserved word. Finally, there may be several process definitions written in the same process form.

```
declaration:
        define pdef (channel ch1, ch2, . . .  chn)  pat1 = exp1  |
               pdef (channel ch1, ch2, . . .  chn)  pat2 = exp2
         and   pdef2 nochannels pat exp;
```

Process Creation

Each instance of a process is created in a *create expression*; the code associated with a process form is applied to the arguments supplied (which must, of course, be of the right type) and a running process is then in existence until the expression in the selected clause yields a result. This newly created process executes concurrently with the process containing the create expression. Although we introduce the notions with a simple, but very typical instance, the full syntax can probably be inferred.

```
create p = pdef exp1 in exp2 end;
```

The syntax is intentionally close to that of local declarations, since the scope of the process identifier, p, is just exp2. pdef is a process definition identifier and exp1 is the argument that will be bound to the formal parameter of the process program. After this binding, process p is active and executes concurrently with the evaluation of exp2 (called the body of the create expression). The value that results from evaluating exp2 is deemed to be the value of this create expression.

Within its scope (exp2 in the above example), a process identifier is taken to be of type α ->> β, where α is the type of the channel record and β to be the type of the result.

It is possible to initiate multiple processes in the one create expression as the following example indicates:-

```
create p1 = pdef1 exp1
and    p2 = pdef2 exp2
in exp3 end;
```

Getting Results

In a create expression, the process identifier's scope is the body (of the create expression) so that it can be used to reference the result of the process (when that result is available), to send messages to the process and as a process descriptor that can be passed to other processes.

There is a polymorphic operator called `result` which takes a process as its argument and yields the result that was produced by that process; `result(p)` will not return anything until process p has terminated.

Message Passing

The way to send a message to another process is by invoking the predefined function *send*:

```
send p #> c exp
```

In this example, p is a process identifier and c is one of the channel names of the process form of which p is a process instance. The construct `p #> c` has type α *channel* and for type consistency the message expression should have type α. The semantics of the expression is that the object that exp evaluates to is sent on channel c to process p. The type of send is α *channel* $\rightarrow \alpha \rightarrow$ *unit*. The sending process is not blocked The only possible exception that can be generated by a send is where the process p has terminated. If process p does not have a channel c, then the error is detected by the type checker.

The messages that are sent to a process on one of its channels are extracted from that channel (a message queue) by the predefined function *get*, an example of which follows:

```
expression:
        get #<c
result:
        x: 'a
```

In this case, c is a channel of the current process, the type of which must have been α *channel*. Both `#>` and `#<` are new operators, chosen to resemble the IO redirection of UNIX.

An Example

The Sieve of Eratosthenes (SOE) is a classic problem with various solutions being algorithms that are capable of efficiently using a large number of processors. In the solution below, we have a pipeline of processes, each one of which takes care of the selection of one prime number. A sequence of all odd numbers which is fed into one end of the array, is filtered as it passes along, so that the sequence that goes to the nth process contains no multiple of any of the first n primes but contains all other members of the original sequence. When the sequence is reduced to nothing a message flows back the other way, gathering primes as it goes.

```
val sieve (channel data:int) ()
  = let prime = get #<data
    in  if prime =-1 then nil
        else create s = sieve ()
             in let fun f -1 = send s#>data -1
                    fun f dv = if dv mod prime <> -1
                               then (send s#>data dv;  f(get #<data))
                               else f(get #<data)
                in f(get #<data);
```

```
            prime::(result s)
            end
        end
    end;
```

The main program is the following function:

```
fun soe(0) = nil  |
    soe(1) = nil  |
    soe(2) = [2]  |
    soe(n) = create s = sieve ()
             in  let fun genlist g = if g<= n
                                     then (send s#>data g;  genlist(g+2))
                                     else send s#>data -1
                 in genlist(3);
                 2::(result s)
                 end
             end;
```

Methodology

There are a number of standard ways of structuring parallel programs, such as worker farms, space partition, data partition etc. We have coded examples of these to show that an application that is amenable to solution by one of these strategies, can readily be written in *paraML* without the programmer worrying about process creation and inter-process communication.

There is insufficient space available to properly discuss the various recipes, so the reader is asked to watch for a subsequent paper.

Implementation

The SML of New Jersey compiler was used as the starting point for the implementation of paraML on the AP1000. It is incomplete at this stage although sufficient of the task is done to have uncovered some interesting problems. These problems and their solutions are presented in [1] and will also be addressed in a forthcoming report.

References

[1] P. Bailey, "*paraML* a parallel extension of ML," B.Sc.(Hons) Thesis, Dept. Comp. Sci, Australian National Univ., (1991).

[2] R. H. Halstead, "MultiLisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems* 7, 4 (October 1985), 501–538.

[3] M. C. Newey, "Towards a CAP Implementation of ML," *Proceedings of 1990 CAP Workshop*, Fujitsu, Kawasaki (Nov. 1990).

Verification of Concurrent Systems in SML *

Paola Inverardi
I.E.I.-C.N.R. Pisa

Corrado Priami
Univ. Pisa, Dip. Informatica

Daniel Yankelevich
Univ. Pisa, Dip. Informatica
HP Labs, Pisa Science Center

Abstract

There can be different views of a concurrent, distributed system, depending on who observes it. The final user may just want to know how the system behaves in terms of its possible sequences of actions, while the designer wants to know which are the sequential components of a system or how it is distributed in space. Moreover, there is no widely accepted semantic model for concurrent systems.

In this paper we describe the use of the SML language in the implementation of a parametric verification tool for process description languages. It allows symbolic execution of processes at different levels, and provides facilities for equivalence checking.

1 Introduction

In this paper we describe the use of the SML language in the implementation of a parametric verification tool for process description languages (PDL). In the recent past, many verification tools for distributed concurrent system, which are based on process description languages, have been proposed (for a detailed survey see [6]). Any of these existing tools is based on a specific theory and, therefore, it is suitable for analisying a particular class of problems. In particular, all the existing tools are based on the so-called *interleaving* models.

The tool we have realized supports more than one concurrency model and allows the user to simply switch from one model to the other. Roughly speaking, it is a parametric tool that permits to observe many aspects of a distributed system. Among the others the temporal ordering of the events and their causal or spatial relation can be studied. An introduction to the general motivations and to the theory underlying our approach is given in [8].

The main motivation underlying our approach is that there is not only one kind of observer of a system. More precisely, there can be different views depending on the kind of observer. The final user of the system may just want to know how the system behaves in terms of its possible sequences of actions, in this respect the interleaving semantics is enough for him. He looks at the system as a black box.

On the other hand, programmers who have to implement systems usually think in a truly concurrent way: two concurrent processes are two concurrent processes not a single non-deterministic one. Also in debugging a big system this way of thinking is more convenient, one *truly concurrent* run of a system gives the same amount of information to a programmer than an exponential number of interleaving computations. Moreover, there is no widely accepted truly concurrent model, but there are many of them. A tool allowing parameterization with respect to various models would also help in understanding which model is more useful and would allow comparisons among them.

Parametrizing a tool in order to catch many different aspects of a distributed concurrent system requires:

- to have a common description language for the various aspects to be considered;
- to have a very fine description of the system at hand, from which all the necessary information for a large number of models can be extracted.

In order to satisfy the two items above we proceed in the following way: we first start from a very concrete representation of the system under specification in terms of a particular transition system (in the sequel denoted by Tccs), whose transitions are labelled by their proofs. On this very detailed description

*Research Partially Supported by Hewlett – Packard, Pisa Science Center

it is then possible to define abstraction functions, called *observation functions*, which permit to recast the chosen observational model from the Tccs structure by throwing away some information on the internal structure of the studied system. As an example, we can define an interleaving observation function which, given a Tccs computation, i.e. a sequence of transitions, returns the computation as observed in the chosen model by forgetting, for any transitions, the whole proof but retaining the label of the performed action. Analogously, it is possible to define functions that permit to retrieve all the common models presented in the literature.

Once the interesting aspects to examine, i.e., once a specific *observation* has been chosen, one of the most widely used facility concerns the ability of proving behavioural equivalences among communicating systems. These equivalences, usually called *bisimulations*, asses that two systems are equivalent if, whenever the first may perform a (possibly complex) activity, the other one may as well, reaching states that are again bisimilar.

According to the purpose of having a very flexible tool for the analysis of distributed concurrent systems, the implementation is modular and open. By open we mean that an expert user can define his/her own observations and equivalences, and then use the standard facilities provided by the tool. We can see the tool as logically divided into two parts: a *kernel* and a *library*. The kernel contains the definitions of the input formalism, the functions to deal with simulation of processes and the functions to deal with the observed behaviour of the processes. Also the interface facilities are contained in the kernel. This part of the tool cannot be modified by the users and it defines also the interface that any observation module has to exhibit. More precisely, each component module of the library must contain the definition of an observation function and of an equality function between the observations of two computations.

To this respect the choice of SML as implementation language has been very convenient since SML allowed an easy design and manipulation of the various modules. Since parametricity and open-endness are two strong requirements for our environment, the modules system of SML turned out to be crucial. On one side, it forces the user to obey to some constraints in defining, from scratch, his/her own component through the notions of signature and structure. On the other side, the user can take advantage of the already defined components by defining a new component as composition of previously defined ones, through the notion of functor. Other important motivations to the use of SML have been the high portability of the language and the high number of existing tools which are implemented in this language. Indeed, a current work is the study of the possibility of the integration of our tool with other existing tools in order to reuse, when possible, already implemented software.

In the following, we describe the architecture of the tool trying to outline the parts which have most benefitted of the use of SML. Then we discuss our experience in using SML.

2 Underlying Theory

In this section, we will briefly recall the main definitions and results of the underlying theory the tool is based on.

An extended explanation of the underlying theory can be found in [3], where a general methodology for the definition of concurrent systems semantics is presented. This methodology is here instantiated with the language CCS [9].

In its general lines, the approach consists of four steps:

1. Define a transition system that captures the operational behaviour of the system. This operational description has to be very concrete, i.e. it has to capture all the information about transitions that the language is intended to describe. It is very natural to associate to CCS a *concrete* transition system, since the only information that a transition can carry on is its own proof. Hence, we choose as the basic level of description for CCS the *proved* transition system, which is the standard transition system of CCS, where the labels describe the proofs of the transitions.

   Proved transition systems have been introduced in [1], and its algebraic versions in [10, 4]. The proved transition system for CCS is here called **Tccs**.

2. Build the computations of the system as paths in the transition system, and structure them as an *observation tree* (order$_{in}$ them by prefix). The *computations* of the transition system Tccs are just sequences of (proofs of) transitions. Once ordered by prefix, the computations an agent can perform generate a (fixed) tree-like structure.

3. Define what are the observations of computations. These appear as labels on the nodes of the observation tree, describing the relevant aspects to be considered.

   To observe a computation means to abstract away some details of the operational description that are hidden to the particular observer. For example, the interleaving observation is defined by abstracting away all the information in the proofs, recording only the actions of each transition and the order in which they are performed. In this way, to each computation is associated a sequence of actions.

   Observational theories of concurrency have been presented in many places, the approach taken here mainly follows [2, 5, 4].

4. Define an equivalence between observation trees, based on the observations defined in step (3).

   The equivalences that can be considered on observation trees are similar to those considered for transition systems. The so-called *bisimulation* equivalences [12] take into account the nondeterministic nature of processes, and hence are very natural for a theory of concurrent systems. Intuitively, two systems are bisimilar if, whenever the first may perform a (possibly complex) activity, the other one may as well, reaching states that are again bisimilar.

   Different bisimulations can be found in the literature, here *weak*, *strong* and *branching* bisimulations are considered. Also trace equivalence is considered, which identifies two processes if they can perform the same sequences of actions.

   In [3] a sound and complete axiomatization of many bisimulations for observation trees has been given. These axioms give the basis for a rewriting-strategy to the bisimulation problem, and they are used in this tool for implementing a rewriting decision procedure for strong bisimulation.

3 The System: architecture and implementation

In this section we describe the logical architecture of our tool. The architecture is illustrated in Figure 1.

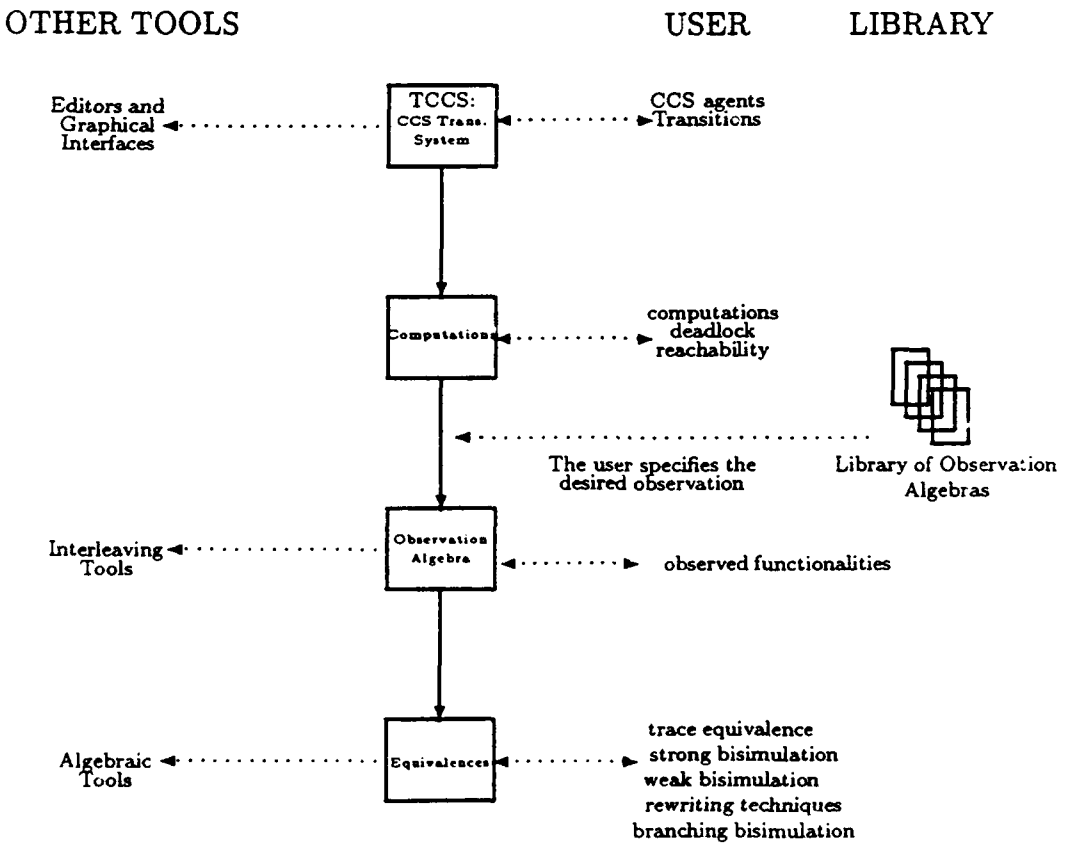


Figure 1. The architecture of the tool

Each box in Figure 1 represents a logical step that a generic user must perform to check the equivalence of two different distributed concurrent systems. Roughly speaking, these boxes are the kernel of our tool. Instead, dashed lines represent possible interactions with other entities like other tools or libraries.

The first step to be performed is to provide the description of the system to be manipulated in CCS. Then, the tool translates the CCS specification in the corresponding Tccs program. Tccs implements our basic syntax and it is the most concrete observation we provide to the user. Since we want to give to the user the possibility of defining his/her own observation algebras, we have equipped this module with a signature. Such a signature says what is exported outside the module, and therefore which are the offered functionalities. We register in the labels of the transitions all the information associated to it, i.e., their proofs in the SOS system defining the operational semantics of the language [13]. Some of the functionalities of this module are to check if a process has deadlock properties and to provide the string representation of processes. Up to now, we use CCS [9], but it is easy to adapt Tccs to support other formalisms by interacting with general compilers of specification formalisms.

The second step concerns the selection of the desired observation from a library. Also in this case, a signature which specifies the general characteristics of observation algebras is provided. The signature specifies which is the structure of agents, transitions and computations after the corresponding structures of the Tccs have been observed by means of abstraction mechanisms. The used abstraction mechanisms are the observation functions which take a Tccs computation and return the corresponding observed computation. All the modules which describe observation algebras must have this signature, therefore the functionalities which appear in this signature are parametric with respect to the chosen algebra. This interface contains all the functions which interpret the constructors of the datatypes agents and transitions. Moreover, the observation algebras must contain functions to perform the sequential composition and the equality of two computations. Note that this is not the syntactical equality function: for instance if the computations are partial orders, it implements an isomorphism checking. Moreover, a function to provide the string representation of observed transitions must be given.

Once the observation algebra has been fixed, the description of the system at hand is translated in a tree structure (the so-called observation tree) which permits to execute equivalence checking. The functionalities provided over observation trees are the equality of two trees up to associativity and commutativity, the building operator starting from an agent, an operator to reduce the size of these trees up to strong bisimulation, and finally a rewriting strategy to test strong equivalence of two trees

Also a functor that lifts the basic functionalities of the tool to the level of observation algebras has been implemented. In other words, it is possible to look at transitions and computations of a system in the particular observation algebra selected. Analogously for the properties like deadlock and reachability. This is a very important module, since it permits to realize the parametricity with respect to the observation. Especially in this part has been crucial the choice of using SML. Indeed, the polimorphic type mechanism of the language has permitted to simply recast the theory in ML thus obtaining the wanted results.

We are now ready to check equivalences. Our tool supports trace equivalence and three bisimulation-based equivalences (strong, weak and branching). All the equivalences are checked by an adaptation of the Paige-Tarajan algorithm based on partition refinement [11]. Finally, also a rewriting strategy to test strong equivalence has been implemented.

Some comments are in order for the library of observations. The user can specify which observation (s)he wants to use in three ways. The first possibility is by choosing an already existent observation from the library. The second one is by defining a new observation algebra from scratch as a new module in SML. This definition follows some standars: it has to fulfill some requirements established in a signature. Once this algebra is defined, the user can choose it from the menu and use the algorithms and facilities with this observation. While giving a definition, previously defined observation algebras and possible reuse of some modules (for example the structure LAB of labels) can be of help to the user. The last possibility is to apply an *operator* to the observation algebras that have already been defined. At present, the operation we allow over observation algebras is the *product*, which takes two observations and compose them in a new one. Obviously, the equivalence obtained in this way is finer than both the equivalences obtained with each observation. Indeed, it is finer or equal than the intersection of the equivalences. Many other operations could be defined over observation algebras, allowing the user to specify an observation domain as an expression in a language of observations. From the implementation point of view this simply means to define a functor which takes two structures which are observations and returns a new observation.

There is a wide spectrum of observations to choose from. The definition of new observations is also easy. In this way, if a new theory is proposed within the methodology presented in [3], it is possible to obtain an automatic support for testing it with few efforts.

The tool is implemented in SML of New Jersey (version 0.66 of 15 September 1990) and runs over UNIX machines. Its code is organized into modules for ensuring a simple debugging and maintenance of the system.

4 Conclusions

Our goal was to implement a prototype of a parametric verification tool for concurrent systems based on a particular theory [7]. The main reason in choosing SML as the implementation language are the following:

- The compiler of NJ-SML is of free-distribution. This makes our tool portable, easy to distribute.
- NJ-SML is an high-level language adequate for rapid prototyping and its type discipline forces a nice style of programming.
- NJ-SML facilitates the interaction with many other existing tools already implemented in SML and permits to reuse code.

Since our main commitment was to stress the parametric aspects of the tool, NJ-SML has been particularly suitable to this purpose because of its algebraic nature. In a parametric setting to have signatures and correspondent structures allows us to establish the basic operation into a signature which can be instantiated with many different modules. Moreover, some of these modules can also be implemented by the user, once fixed the signature. The *functors* facility provides a very powerful way of implementing parametric modules with respect to an underlying algebra.

Besides these advantages, the fact of having non-functional constructs has been very useful since our system has a part which is algebraic in nature and also includes some imperative algorithms [11].

However, NJ-SML interpreter and compiler lacks of some important features. The version we have used is not well-documented and it does not provide facilities for developing interfaces, graphics and so on. It is also difficult to modify system parameters. Moreover, the error handling routine is poor. We think that all drawbacks are due to the absence of an integrated programming environment, which will be very helpful in prototyping in SML. A minor drawback is caused by the absence of overloading facilities which forced us to use too many different identifiers.

Acknowledgment: Giovanni Mandorino has implemented the algorithms for bisimulation testing and has participated in many discussions.

References

[1] G. Boudol and I. Castellani. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informaticae*, 11(4):433–452, 1988.

[2] P. Degano, R. De Nicola, and U. Montanari. CCS is an (augmented) contact free C/E system. In M. Venturini Zilli, editor, *Advanced School on Mathematical Models for the Semantics of Parallelism, 1986*, volume 280 of *Lecture Notes in Computer Science*, pages 144–165. Springer-Verlag, 1987.

[3] P. Degano, R. De Nicola, and U. Montanari. Universal axioms for bisimulation, 1991. Submitted for publication. An extended abstract appeared in Proc. Workshop on Concurrency and Compositionality, Goslar, 1991.

[4] G. Ferrari. *Unifying Models of Concurrency*. PhD thesis, TD-4/90, Dipartimento di Informatica, Università di Pisa, 1990.

[5] G. Ferrari, R. Gorrieri, and U. Montanari. An extended expansion theorem. In *Proceedings TAPSOFT '91*, volume 431 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1991.

[6] P. Inverardi and C. Priami. Evaluation of tools for the analysis of communicating systems, 1991. Bullettin of EATCS, No.45.

[7] P. Inverardi, C. Priami, and D. Yankelevich. A parametric verification tool for distributed concurrent systems, 1992. Submitted to CAV'92 - Montreal.

[8] P. Inverardi and D. Yankelevich. Parametric true concurrent reasoning about distributed systems. Technical Memo HPL-PSC-91-24, Hewlett-Packard Laboratories, Pisa Science Center, Pisa, 1991.

[9] R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.

[10] U. Montanari and D.N. Yankelevich. An algebraic view of interleaving and distributed operational semantics for ccs. In *Proceedings Category Theory and Computer Science '89*, volume 389 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 1991.

[11] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[12] D. Park. Concurrency and automata on infinite sequences. In 5^{th} *GI Conference*, vol. 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981.

[13] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

5 Appendix: A Very Simple Example

The input of the user is *italic* while plain text denotes the tool output. Comments are in *cursive style*.

The system displays a menu of observations. The user has to choose which observation (s)he wants to use.

CHOOSE AN OBSERVATION ALGEBRA:
SPO $\Longrightarrow$Spatial Partial Orders Strong
SPW $\Longrightarrow$Spatial Partial Orders Weak
POR $\Longrightarrow$Partial Orders Strong
POW $\Longrightarrow$Partial Orders Weak
INT $\Longrightarrow$Interleaving Strong
WEA $\Longrightarrow$Interleaving Weak
STE $\Longrightarrow$Multisets Strong
ALS $\Longrightarrow$Abstract Localities Strong
...

POW

Once the observation algebra has been chosen, the system displays a menu with the bisimulation algorithms.

CHOOSE THE BISIMULATION ALGORITHM:
S $\Longrightarrow$Strong Congruence
W $\Longrightarrow$Weak Equivalence and Congruence
B $\Longrightarrow$Branching Equivalence and Congruence
T $\Longrightarrow$Trace Equivalence
R $\Longrightarrow$Strong Rewriting

W

The system is now ready in a Weak Partial Orders setting. The user can now specify different agents.

val $p = \alpha.\beta + \beta.\alpha$;
val $q = \alpha|\beta$;
test(p,q);
$\alpha.\beta + \beta.\alpha$ and $\alpha|\beta$ are not WEAKLY bisimilar
cho();
The user asks to change the observation.

...

WEA

The Interleaving Weak algebra is chosen.

test(p,q);
$\alpha.\beta + \beta.\alpha$ and $\alpha|\beta$ are WEAKLY bisimilar and WEAKLY congruent

A File System in Standard ML

Drew Dean
Drew.Dean@cs.cmu.edu

April 13, 1992

Abstract

This paper discusses the design and implementation of a distributed file system in Standard ML. The file system is implemented using persistent storage from the Carnegie Mellon Venari project, and communicates *via* Mach Remote Procedure Calls. This is work in progress, so details are subject to change, and no benchmarks are available yet. Performance figures will be available for the workshop.

1. Introduction

Many file systems have been implemented since the introduction of magnetic disk storage. To better organize files for humans, hierarchical directory structures are generally used. The implementation of these file systems has traditionally been done in relatively low level languages such as C. In Berkeley Unix,[1] the user sees a single directory structure. Under the covers, however, Unix is dealing with disk blocks as untyped arrays of bytes. This is difficult and leads to code such as:

```
#define itod(fs, x) \
        ((daddr_t)(cgimin(fs, itog(fs, x)) + \
        (blkstofrags((fs), (((x) % (fs)->fs_ipg) / INOPB(fs))))))
```
[2]

After implementing such structures on disk, more code must be written to allow network access. In contrast, Standard ML permits a concise implementation of a very similar abstraction using ML datatypes, persistent storage, and Mach RPC to communicate. The ML implementation does not need to worry about superblocks, inodes, or cylinder groups — it merely sees one large tree to traverse. Since tree traversal is one of ML's strengths, the result is much more elegant than the above C example. The directory structure contains two mutually recursive types, `File` and `Dirent`. The root directory of a file system is initialized to `DIRECTORY(ref [] : Dirent list ref)`, and so is a `File`.

```
datatype File = DATA of ByteArray.bytearray ref
              | DIRECTORY of Dirent list ref
              | LINK of string
and Dirent = FILE of string * int * time_t * File
```

The string in the `Dirent` is the file name, as expected. The integer is a unique ID assigned to the file at creation.

---

[1] Unix is a registered trademark of Unix System Laboratories, Inc.

[2] From `/usr/include/sys/fs.h` on a Mach 2.5 system.

2. Network Overview

In the 1990s, there is little point in designing a file system without taking network access into account. Since this is ML, a nearly stateless protocol is indicated. Inasmuch as there are only a few operations on a file, any stateless protocol will resemble Sun's NFS [2] to some extent. For now, we adopt an NFS-like protocol, with the major difference being that our protocol communicates via Mach RPC instead of Sun RPC. With that said, the major RPCs and an informal definition of their semantics are as follows:

- **Lookup** takes a pathname and returns as a file handle the unique identifier for the file.
- **Read** takes a file handle, offset, and request size, and returns a string, along with the number of bytes read.
- **Write** takes a file handle, offset, request size, and data as a string, and returns the number of bytes it was able to write.
- **Mkdir** takes a pathname and creates an empty directory of that name if possible.
- **Rmdir** removes the empty directory named by its argument.
- **Create** creates a zero-length file named by its argument, if possible.
- **Unlink** removes its argument, if it does not name a directory, from the file system.
- **Makelink** makes its second argument a symbolic link to its first argument.
- **Readlink** reads the contents of the symbolic link whose pathname is passed to it.
- **Stat** returns the unique ID, time of creation, and file type of the pathname passed to it.
- **Readdir** takes a file handle, (which must point to a **DIRECTORY** file), and a magic cookie. It returns the directory entry corresponding to the cookie, and a new cookie, for the next directory entry. The cookie is implemented by using the unique IDs; the first file in a directory is read by passing in a special cookie. The same special cookie is returned to indicate the end of the directory.

Pathnames are passed as a list of pathname components; it is assumed that all but the last are directories, and the last component is a file or directory name. Since symbolic links are resolved by the client, they are stored as single strings. Since the actual definition of this interface is in MIG[3], both C and SML clients can use it. A C interface, exporting Unix-like **open**, **read**, **write**, **lseek**, **mkdir**, **rmdir**, **symlink**, **readlink**, **stat**, and **close** functions would be an interesting exercise, especially for benchmarking purposes.

All calls handle symbolic links by returning an error code to the client, along with the number of the first pathname component that is a link. The client then does a **readlink**, to resolve the link, and retry the operation with the result. Since symbolic links should be resolved from the client's viewpoint (in the case of multiple file systems), an interaction of this sort is needed. It would be fairly simple to optimize this interface by having the operation return the contents of the link along with the error code, thus avoiding an immediate **readlink** RPC. It is assumed that the client keeps a table of mounted file systems and their mount points; the client can find the longest pathname prefix to determine the correct file system. The client is responsible for loop detection. The client keeps a list of links it has traversed, and linearly searches this list for duplicates. While this approach may seem to have performance problems, based on my study of symbolic link usage on various Unix machines, links are relatively rare, and the vast majority of links point to their final targets. Thus, in most cases, link resolution takes constant time, while the worst case is $O(n^2)$, where n is the number of intermediate links.

[3] MIG is the Mach Interface Generator. It generates stubs for using Mach's Remote Procedure Calls. It is roughly analogous to Sun's **rpcgen**. While MIG originally only generated stubs for C and C++, Frederick Knabe and I reimplemented it in Standard ML to produce Standard ML.

After translating the above RPC definitions into MIG format, SMLMIG produces the following signature for the implementation of the file server:

```
signature MLFS =
    sig
        exception mlfs of int
        val lookup : MachIPC.port * string list -> int * int
        val read :  MachIPC.port * int * int * int -> int * string * int
        val write : MachIPC.port * int * int * int * string -> int * int
        val mkdir : MachIPC.port * string list -> int
        val rmdir : MachIPC.port * string list -> int
        val create : MachIPC.port * string list -> int
        val unlink : MachIPC.port * string list -> int
        val stat :   MachIPC.port * string list -> int * int * int * int
        val readlink : MachIPC.port * string list -> string * int
        val makelink : MachIPC.port * string list * string -> int
        val readdir : MachIPC.port * int * int -> string * int * int
    end
```

Note that the last `int` return value is an error code, and the `string lists` are pathnames. The `MachIPC.port` is the fileserver's Mach port.

The client interface has two parts: one part executes the client half of the protocol, and the other is an application interface. The MIG-generated client interface is essentially the same as the server interface. The application interface wraps the RPCs in logic to handle symbolic links and errors, resulting in a Unix-like interface, but using exceptions to report errors instead of return codes and a global variable.

```
signature MLFSCLIENT =
sig
    val pathSep : string;
    structure Common : COMMON
    type fh
    val fopen    : string -> fh
    val read     : fh * int * int -> int * string
    val write    : fh * int * int * string -> int
    val mkdir    : string -> unit
    val rmdir    : string -> unit
    val create   : string -> unit
    val unlink   : string -> unit
    val stat     : string -> int * int * Common.FileType
    val readlink : string -> string
    val symlink  : string -> string -> unit
    val readdir  : fh -> int -> string * int
    val reinit   : unit -> unit
    val mount    : string -> string -> unit
    val umount   : string -> unit
end
```

3. Implementation Details

The implementation of the unique ID scheme implies that a file system cannot have more than 2^{31} files. This is not considered to be a problem. A more natural unique ID would be nice, but there are no obvious candidates, since nothing is fixed in the file system. The file system is strictly a tree; Unix-style hard links do not exist. The absence of hard links is not considered a major drawback, since symbolic links are an effective replacement. While an inode-like structure could be used, it doesn't seem to fit well into a high-level model

of a file system. If the lack of inode-like structures becomes a problem, they would be relatively simple to implement and would not require a complete rewrite of the file system. The major changes that this would imply are the elimination of the separate unique IDs, and `Dirent`'s would have an inode component instead of a `File` component. Presently, `time_t` is an integer, used to store the file creation date. Symbolic links are not interpreted by the server, so they are stored as simple strings, presumably parseable by the client.

The allocation system for the file system is one of its more interesting features. Files are allocated sequentially, until the heap fills up. At this point, the heap is garbage collected. Since the garbage collector is a copying collector, this corresponds to disk defragmentation in a traditional file system. A generational strategy for garbage collection will be very effective, as file lifetimes tend to be either very short or very long — consider, for example, temporary files made by compilers and text editors, vs operating system binaries and mailbox files. It may be necessary to periodically run the garbage collector, as a full scan of a large heap is bound to be slow. Along with the lack of inode-like structures, this allocation scheme is an experiment that may or may not succeed.

4. Low Level Details

The unique ID scheme uses a simple hash table that is updated by the `create`, `mkdir`, `unlink`, `rmdir` RPC's, and used by the `lookup` RPC. Since unique IDs are consecutive integers, the identity function is a perfect hash function, modulo hash table size. The hash table resolves collisions by chaining; this was very simple to implement in ML.

The persistence mechanism for the file system is based on the Venari project's work. In turn, this code is based on Mashburn and Satyanarayanan's Recoverable Virtual Memory (RVM) work [4]. The Venari persistence work exposes two separate heaps to the ML programmer: the traditional volatile heap, and the persistent heap[3]. Here the persistent heap contains the file system and little else, while the volatile heap serves SML/NJ's needs for runtime storage. A key performance issue for persistence is how much data is written out to disk. Since ByteArrays cannot be extended, writing past end-of-file requires that a new, larger ByteArray be created, data copied, and new data written. The current system does not allocate any space on file creation, but allocates 1024 bytes on the first write to a file. After that, the files grow exponentially when writing past end of file. Directories were made to contain references to the list of directory entries to minimize the amount of directory copying necessary.

Once the file system itself has reasonable performance, it is time to turn attention to network performance. In general, of course, large transfers traverse the network faster than small transfers, just as with disk transfers. The problem, however, comes when IP has to fragment packets. NFS, by default, sends and receives 8 KB blocks. Since Ethernet has a maximum packet size of 1500 bytes, each NFS transmission becomes 6 fragments. While this is acceptable on a single Ethernet, NFS does not work well across bridges or routers, where losing one fragment is fairly common. Reducing the transfer size to fit within 1 network packet provides a substantial performance increase in this case. While MTU (Maximum Transmission Unit) discovery would be the right thing to do, it is very difficult in general. Since IP requires networks to have a minimum MTU of 576 bytes, picking a transmission size of 1024 bytes seems fairly safe. In the common case of routing between Ethernets, there will not be any fragmentation. In the worst case, there will be at most two fragments, so there should be a reasonable chance (certainly much better than with six fragments) of all the fragments getting through the bridge/router. Note that FDDI has a MTU of about 4500 bytes, and in general, increasing network speed seems to bring increasing MTUs, so while a RPC size smaller than 1500 bytes will become less optimal, the performance problems of packet fragmentation are always avoided. Since the page size of most modern workstations is in the 4–8 KB range, a 1 KB network transmission should give a nice readahead effect, assuming the file is being read sequentially from start to end, which is extremely common.

5. Future Work

One of the key benefits of this file system implementation is its brevity. The entire file system is less than 1000 lines of user written code, so it is ideal for performing file system experiments. Tuning the file system is the first priority:

- The best base for the exponential growth of the ByteArrays can be found experimentally. The tradeoff between byte copying cost and wasted space needs to be determined, and perhaps made adjustable on a per file system basis.
- An alternative scheme to support growable files would use a list of fixed length ByteArrays. While this could improve write speed significantly, read performance may suffer slightly due to the need to handle reads that straddle the block boundaries imposed by the server. Since reads tend to dominate writes, this is an interesting tradeoff.
- Pathname-to-handle lookups should be cached. This cache can have a hit rate of 70–80%.[1]
- The file server could be multi-threaded. If a slight decrease in reliability is tolerable, one thread could update persistent storage every 30 seconds, like the Unix `update` daemon. This could dramatically decrease the number of disk writes actually done by the file system for processes appending to the end of large files. The file server could also fork a thread (up to a limit) for each incoming request, in which case a multi-threaded implementation would change the `Dirent` type to add a lock per file. Depending on the desired semantics of multiple concurrent writers to a file, locks might only be used on directories. Locking would proceed with each thread holding two locks at a time, much like a monkey climbing hand-over-hand up a tree.

The major performance issue not addressed here is disk seeks. In general, controlling data placement on the disk is impossible from ML. However, it is worth noting that RVM uses a linear mapping from virtual memory address to disk block, so if it were possible to influence ML's memory use, then some control over disk layout would become possible.

After the file system is properly tuned, there are several interesting directions to take this research. One would be to follow the ideas of Gifford in [5], and make a content-addressable file system. With current RPC technology, search engines would have to be implemented entirely on the server side. The `grep` family and the Venari signature matcher could be integrated into the server without undue trouble, and one new RPC, `mkdirbycontent`, added.

The file system currently has no protection mechanisms. While Unix-style mode bits are useful to protect users from themselves, simple protection schemes do not provide any real security across networks. I believe it is better to explicitly offer no security rather than poor security – this way there are no nasty surprises, other than human error. The design of the file system does not preclude any access controls; again the conciseness of implementation favors experimentation with new authentication protocols.

Another interesting direction to explore would be to add transactions to the file system. Transactions could be both used to structure the file server, and made user-visible for users desiring consistent updates of multiple threads. The Venari project is currently investigating the semantics of transactions in ML.

6. ML Issues

The design and implementation of the file system raised two points about the ML language: there is little control over memory use, and there is no nice way to specify when non-pure functions (that is, functions that generate and/or observe side-effects) should be evaluated. For most application programs, memory usage isn't particularly important, but for systems programming it is vital. For example, if a cache is about to be

paged out to disk, it may make more sense to discard the cache instead. Consider a file system cache: there is no reason to write the cache out, as it is almost as fast to read the original blocks back in again. Standard ML does not offer this level of control, which would be quite useful inside an operating system.

The following fragment illustrates the function-application time problem:

```
functor Test(structure Netname : NETNAME
             structure RPC : RPC) =
struct
        val file_server_port = Netname.lookup("localhost", "ML_File_Server")
        :
        fun lookup filename = RPC.lookup(file_server_port, filename)
end
structure T = Test(Netname)
```

Assume `RPC` is a structure containing client-side interfaces to remote procedure calls and `Netname` is the interface to a network name server, from which processes can check in or out interprocess communication ports. Clearly, `T.file_server_port` should be bound to the port the file server is listening on at runtime, not at functor application time, as presently happens. While the desired functionality can be simulated by judicious use of references, the result is not very elegant. With careful thought, it should be possible to provide this functionality without too much modification to the semantics of ML.

7. Summary

With Standard ML, SMLMIG, and the Venari persistence implementation, it is possible to implement a distributed file system. The implementation is concise and has reasonable performance. A variant of a common protocol is used for access to the file server, while the file server itself is implemented in a fairly unorthodox fashion.

I'd like to thank Eric Cooper, Frederick Knabe, Gene Rollins, Jeannette Wing, and the entire Venari project for making this work possible, and for many helpful suggestions about this paper.

References

[1] Leffler, Samuel J., et al. *The Design and Implementation of the 4.3 BSD Unix Operating System*, Addison-Wesley, 1989.

[2] Sun Microsystems, Inc. *NFS: Network File System Protocol Specification*, March 1989, RFC-1094, SRI International.

[3] Nettles, Scott M., and Jeannette Wing, *Persistence + Undoability = Transactions*, August 1991, CMU-CS-91-173.

[4] Mashburn, Henry, and M. Satyanarayanan, *RVM: Recoverable Virtual Memory*, Version 0.1, March 1991.

[5] Gifford, David K., and James W. O' Toole, *Intelligent File Systems for Object Repositories*, in LNCS 563, Springer-Verlag, 1991.

Implementing a Mixed Constructive Logic in Standard ML

James T. Sasaki
Computer Science Department
University of Maryland Baltimore County
Baltimore, MD 21228
sasaki@umbc4.umbc.edu

Ryan Stansifer
Department of Computer Sciences
University of North Texas
Denton, TX 76203
ryan@ponder.csci.unt.edu

April 10, 1992

1 Introduction

In this paper we describe an implementation of a programming language MCL which is based both on constructive and classical logic. The language is intended for specifying, constructing, and executing verified programs. By combining elements of both constructive and classical logic, it is possible to eliminate some computational inefficiencies found in purely constructive type theories, but in a way that does not drastically change the style of proofs. The availability of both constructive and classical operations makes it possible to write specifications not available in other systems. We are working on a prototype implementation of the language in Standard ML [5].

2 Constructive type theory

In a Constructive Type Theory (CTT), proofs of constructive logic can be viewed as programs [2, 4, 7]. If F is a logical formula in a CTT, then a constructive proof π of F is a computation that stands for a *justification*, a piece of evidence sufficient for believing in the truth of F. In the same way that an arithmetic expression stands for an element of datatype `int`, the proof π stands for an element of datatype F. Thus in a CTT, proofs are programs, and formulas are datatypes.

Datatypes in Pascal, say, serve as very crude specifications of the programs that compute elements of the type. Formulas in CTT's serve as more interesting specifications. For example, formula (1) below could serve as the specification of dividing x by y.

$$\forall x : \mathtt{int} \,.\, \forall y : \mathtt{int} \,.\, \exists q : \mathtt{int} \,.\, \exists r : \mathtt{int} \,.\, x = q * y + r \tag{1}$$

For the purposes of this paper we are not concerned about specifying more completely the relationship between x, y, q, and r, although this is certainly possible. The two existential quantifiers here will suffice to demonstrate a source of computational inefficiency. In MCL syntax, formula (1) is written

```
all x:Int.all y:Int.some q:Int.some r:Int.x=q*y+r
```

As a datatype, a universal formula acts like a function space, and an existential formula acts like a product. Hence an expression that has the datatype given by formula (1) is a curried function that takes two integers (call them x and y) and returns a nested tuple with two integer components q

and r and a final component of type $x = q * y + r$ that specifies the relationship between x, y, q, and r.

Since proofs are programs, one can write translators for constructive logic. The basic idea of taking a proof of a formula and (say) interpreting it to produce a piece of evidence is the same as the basic idea of taking an expression and evaluating it to get a value.

Unfortunately, the most obvious way to implement a CTT causes runtime inefficiencies because uninteresting pieces of evidence are calculated. Consider the following proof/program of (1) written in MCL:

```
fun x: Int . fun y: Int .
    <x Div y, <x Rem y, e> into some r: Int. P> into some q: Int. Q
```

In this proof we use some-introduction twice to exhibit the quotient and remainder. The expression e stands for the proof that the chosen quotient and remainder have the right properties. P and Q abbreviate the appropriate bodies of the existentials. The type-checker needs e to verify that the program meets the specification (1). At runtime, if the function above is evaluated for specific x and y, then values for $x \,\mathtt{div}\, y$, $x \,\mathtt{rem}\, y$, and e will be calculated. If we are not interested in the actual piece of evidence denoted by e (and this is the usual case), then the evaluation of e is wasteful. For our particular example, this is not much of a problem, but in the general case, evaluation of a function that justifies $\forall x : t \,.\, \exists y : t' \,.\, F$ could spend an arbitrary amount of resource calculating a justification for F.

The usual method [3, 6, 7, 9] for avoiding this problem involves modifying the specifications. Find the formulas whose proofs would cause undesired computations, then replace those formulas by formulas that are classically equivalent but have only computationally uninteresting proofs. (The proofs are uninteresting in the same sense that non-`()` expressions of type `unit` in SML are uninteresting.) Modify the implementation so that it detects and optimizes away these uninteresting proofs. The result is more efficient execution.

Unfortunately, this technique has the side effect of making these more efficient proofs harder to read. For example, take the specification (1) above. If all we are interested in is q given x and y, we could rewrite (1) as follows:

$$\forall x : \mathtt{int} \,.\, \forall y : \mathtt{int} \,.\, \exists q : \mathtt{int} \,.\, \neg \forall r : \mathtt{int} \,.\, \neg (x = q * y + r) \tag{2}$$

Proofs of (2) are classically equivalent to proofs of (1), but such proofs are also less clear.

3 Mixed constructive type theory

A more direct way to improve the efficiency of the interpreter is to allow the programmer to indicate what parts of the program are computationally significant. This is like Hoare logic in which the program segments, which indicate the computational part, are entirely distinct from the assertions, which are important for the purposes of verification. For example, the assertions can refer to variables that do not appear in the program and have no meaning at runtime.

In MCL we are exploring a language in which the user specifies the computational content by using both constructive *and* classical quantifiers and operations in specifications. We keep $\forall$, $\exists$, and $\vee$ with their constructive meaning, but now add $\forall^*$, $\exists^*$, and $|$ as the classical counterparts. Now we can write the following specification:

$$\forall x : \mathtt{int} \,.\, \forall y : \mathtt{int} \,.\, \exists q : \mathtt{int} \,.\, \exists^* r : \mathtt{int} \,.\, (x = q * y + r) \tag{3}$$

which is computationally equivalent to (2) when uninteresting proofs of (2) are removed.

Programs in this logic can have two parts, a constructive part that actually denotes computations to be done, and a classical part that discusses but does not actually *do* computations. The rules of constructive logic are mimicked in the classical parts of the logic, but the classical parts of the proof also have access to rules of logic that are not constructively true. For example, to prove either $\exists r : \mathtt{int} \,.\, F$ or $\exists^* r : \mathtt{int} \,.\, F$, you have to show how to calculate an appropriate r, but in the constructive version, this calculation must be computable; in the classical version, this calculation does not have to be computable.

MCL is based on the mixed constructive logic S1 of [8] where a formal semantics can be found. The mix of classical and constructive formulas in S1 makes it possible to write the formulas that are expressible when we mimic classical logic within constructive logic. Furthermore, it is also possible to write certain formulas that are otherwise inexpressible. For example, the formula $(\forall^* x : t \,.\, \exists y : t' \,.\, F) \Rightarrow (\exists y : t' \,.\, \forall^* x : t \,.\, F)$ is true in the semantics of [8], but is not known to be expressible in, for instance, NuPrl [2].

The rest of the paper is devoted to a description of MCL. We will concentrate mainly on syntactic and implementation issues here.

4 Language

In this section we give a brief overview of the language. The most important constituents of the language are the expressions/proofs and the types/formulas. We give a portion of the grammar for each of these two categories to suggest the basic appearance of the language.

Expressions consist of basic computational units like integer numerals, pairs and functions, as well as laws of logical deduction. Among the latter are the constructs for proving that something exists (some introduction), and the construct for reasoning about the existence of something (some elimination). All introduction and all elimination correspond to function abstraction and function application, respectively.

$$
\begin{aligned}
\langle expr \rangle \;::=\; & \langle ident \rangle \\
::=\; & \langle function\ symbol \rangle \\
::=\; & \mathtt{<}\langle expr \rangle \mathtt{,} \langle expr \rangle \mathtt{>} \\
::=\; & \langle expr \rangle \mathtt{.1} \\
::=\; & \langle expr \rangle \mathtt{.2} \\
::=\; & \mathtt{injl}\ \langle expr \rangle\ \mathtt{:}\ \langle type \rangle\ \mathtt{end} \\
::=\; & \mathtt{injr}\ \langle expr \rangle\ \mathtt{:}\ \langle type \rangle\ \mathtt{end} \\
::=\; & \mathtt{case}\ \langle expr \rangle\ \mathtt{when}\ \langle ident \rangle\ \mathtt{then}\ \langle expr \rangle\ \mathtt{when}\ \langle ident \rangle\ \mathtt{then}\ \langle expr \rangle\ \mathtt{end} \\
::=\; & \mathtt{fun}\ \langle ident \rangle\ \mathtt{:}\ \langle type \rangle\ \mathtt{.}\ \langle expr \rangle \\
::=\; & \langle expr \rangle\, \langle expr \rangle \\
::=\; & \mathtt{<}\langle expr \rangle \mathtt{,} \langle expr \rangle \mathtt{>}\ \mathtt{into\ some}\ \langle ident \rangle\ \mathtt{:}\ \langle type \rangle\ \mathtt{.}\ \langle type \rangle \\
::=\; & \mathtt{let}\ \mathtt{<}\langle ident \rangle \mathtt{,} \langle ident \rangle \mathtt{>}\ \mathtt{:}\ \langle type \rangle\ \mathtt{:=}\ \langle expr \rangle\ \mathtt{in}\ \langle expr \rangle\ \mathtt{end}
\end{aligned}
$$

These are not all the expressions in the language. In particular, certain acts of reasoning about integers, equality, and the like must also be represented by expressions in the language. For example, the expression $\mathtt{ax2}(e_1, e_2)$ is an element of a certain equality type.

The collection of types in MCL contains primitive types like Int, but most important, types corresponding to the logical connectives conjunction, disjunction, and implication. Conjunction

plays the role of the type of pairs. A more realistic programming language would contain more general types for tuples and records, but they are not necessary for the purposes of studying the efficiency of languages based on CTT's. Implication subsumes the role of the type of functions. It is actually a special case of the "all" type. Quantifiers have both a classical and constructive form.

⟨*type*⟩ ::= ⟨*ident*⟩
::= ⟨*primitive type*⟩
::= ⟨*expr*⟩ = ⟨*expr*⟩
::= ⟨*type*⟩ -> ⟨*type*⟩
::= ⟨*type*⟩ | ⟨*type*⟩
::= ⟨*type*⟩ \/ ⟨*type*⟩
::= ⟨*type*⟩ & ⟨*type*⟩
::= all ⟨*ident*⟩ : ⟨*type*⟩ . ⟨*type*⟩
::= all* ⟨*ident*⟩ : ⟨*type*⟩ . ⟨*type*⟩
::= some ⟨*ident*⟩ : ⟨*type*⟩ . ⟨*type*⟩
::= some* ⟨*ident*⟩ : ⟨*type*⟩ . ⟨*type*⟩

As in the language F-sub [1] there is a special type ?. The symbol ? can be used in place of a type in hopes that the MCL system will be able to determine the correct type.

Here is the remainder example in MCL.

```
(fun x:Int. (fun y:Int.
  <(Div <x,y>), <(Rem <x,y>), (ax2 x,y)>
    into (some r:Int.(x=(Add <(Mul <q,y>),r>)))>
      into (some q:Int.(some r:Int. (some r:Int.(x=(Add <(Mul <q,y>),r>))) ))
))
```

Here is the same example again. This time the MCL system is able to fill in some of the type information.

```
(fun x:Int. (fun y:Int.
  <(Div <x,y>), <(Rem <x,y>), (ax2 x,y)> into (some r:Int.?)>
    into (some q:Int.?)))
```

5 Implementation

In this section we describe the SML implementation of the MCL language. This basic approach could be taken to implement any interactive language system and is instructive for that reason. We show the top-level structure of the MCL system and show how SML can be used to make a stand-alone system.

The function `TopLevel` does the so-called read/eval/print loop. In fact, it does six things.

1. Prints a prompt to cue the reader for input and reads a line of input.
2. Parses the input line. This is the function of `parse`. This creates an abstract syntax tree `ast`.

3. Analyzes the expression. In particular, checks the types of all the subexpressions. The function `type_check` performs this part.

4. Evaluates the expression to normal form.

5. Prints the expression, its type, and its normal form.

6. Loops.

Here is a (slightly simplified) version of the top-level function.

```
fun TopLevel () = (
    let
      val input = read ();
    in
      let
        val ast = parse (input);
        val (e',tau) = type_check (InitialTypeEnv, ast);
        val e'' = eval (InitialEnv, e');
      in
        output (std_out, "expr:  " ^ (print e') ^ "\n");
        output (std_out, "type:  " ^ (print_type tau) ^ "\n");
        output (std_out, "eval:  " ^ (print e'')) ^ "\n");
        flush_out std_out
      end handle e =>
        output (std_out, "Exception raised:  "^(System.exn_name e)^"\n");
    end;
    TopLevel ()                                  (*  Loop  *)
)
```

Notice that exceptions during `read`, in particular interrupts, are not captured to allow the function `TopLevel` to be exited.

The parser was implemented using ML-Lex and ML-Yacc, a lexical analyzer generator and parser generator for Standard ML written by Andrew Appel, James Mattson and David Tarditi of Princeton University. This is a particularly convenient way to build elements of the appropriate datatypes. In this way full advantage can be taken of the ML language in representing and manipulating syntactic terms of a language without sacrificing some user-friendly concrete syntax. Although a yacc parser is not as simple as some LISP read macros, it is flexible, general, and based on well-understood concepts.

It is possible in Standard ML of NJ to create a stand-along interpreter for a language by using the command **exportML**. The following code causes an executable image of the ML system to be written to the file `mcl`.

```
if exportML "mcl"
  then (
    output (std_out, "MCL version 0.4\n");
    output (std_out, "  Type ^C to leave MCL and get back to SML/NJ\n");
    TopLevel ()
  ) else ();
```

The function **exportML** returns false when the image is created. The executable image is a snapshot of the system as of the time the function was called. Thus the image can be preloaded with all the functions required for MCL by defining them before the execution of the **exportML**. Executing the file **mcl** causes the ML system to be reactivated at the point where the image was created. However, **exportML** returns the boolean value true in the reactivated system. Hence, using the code above starts the top-level read/eval/print loop as soon as the saved image is executed.

References

[1] Luca Cardelli. *F-sub, the System.* Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, February 1992.

[2] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. Rance Cleaveland, J. F. Cremer, Robert W. Harper, Jr., D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden James T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NUPRL Proof Development System.* Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[3] S. Hayashi and H. Nakano. *PX: A Computational Logic.* MIT Press, Cambridge, Massachusetts, 1988.

[4] Per Martin-Löf. Constructive mathematics and computer programming. In L. J. Cohen, J. Los, H. Pfeiffer, and K.-P. Podewski, editors, *Proceedings of the Sixth International Congress for Logic, Methodology and Philosophy of Science, Hannover, 22-29 August 1979*, pages 153–175, Amsterdam, 1982. North-Holland.

[5] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[6] C. Mohring-Paulin. Extracting F_ω programs from proofs in the calculus of constructions. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 89–104, January 1989.

[7] James T. Sasaki. *Extracting efficient programs from constructive proofs.* PhD thesis, Department of Computer Science, Cornell University, 1986.

[8] James T. Sasaki. Verified programs in a simple type theory with mixed constructivity. Technical Report TR CS-91-20, Computer Science Department, University of Maryland Baltimore County, 1991.

[9] Anne Sjerp Troelstra. Aspects of constructive mathematics. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 973–1052, Amsterdam, 1977. North-Holland.

Experiences with ML for Building an AI Planning Toolbox

Tom Gordon Joachim Hertzberg Alexander Horz

GMD, AI Division,
Schloß Birlinghoven,
D-5205 Sankt Augustin 1, F.R.G.
e-mail: {thomas, hertz, horz}@gmdzi.gmd.de

Abstract

The qwertz toolbox is a system of ML modules for implementing AI planners. We discuss why ML was chosen to implement the system, provide a few guidelines for using the language, based on our experience, and make some suggestions for language extensions and programming environments.

1 Background

The topic of our research is AI planning [1], i.e., constructing a course of action for an agent, given information about the current state of affairs and the agent's goals. Although planning work in general, and of course our own results (e.g., [3, 7]) are interesting, they will not be of further concern to us here. Rather, we focus on the reasons for our decision to implement qwertz, a planning software toolbox, in Standard ML, and make a few comments about the language based on our three years of practical experience with it.

The reason behind our choice of ML has to do with the state of the art in planning at the project start time in 1988. In those days, many researchers, including ourselves, felt the field suffered from a serious problem: there was an extensive literature about algorithms, systems, and techniques—such as nonlinearity, abstraction, or hierarchical planning—but this literature was typically insufficiently precise and complete to enable others to easily reimplement what was described. To quote T. Dean from a paper written then [6]:

> Today, these basic and intuitively appealing methods (for classical AI planning) are known largely by association with archaic and poorly understood programs; these methods cry out for careful reexamination and precise formulation.

One of our project goals was to rectify this situation. We aimed to "rationally reconstruct" classical planners such as to facilitate their reimplementation by others, in their favorite language. Thus, clear design and readable code were of prime importance, rather than tricky efficiency. Also, we dreamed of finding a clear conceptual organization of planning methods which could be mirrored by an analogous modularization of our planning software: orthogonal methods should be implemented orthogonal modules (unless the methods only *appear* orthogonal).

So, setting aside the particular scientific problems of AI planning, we faced two general software engineering issues. First, architecture: How to build a toolbox which faciliates the combination of

orthogonal methods, and experimentation with alternative implementations of each method? And second, didactics: How to avoid the trap of understanding these methods ourselves, but failing to deliver a description of them sufficient to allow others to easily reproduce our results? To cope with these problems, we decided to use three techniques:

Functional Programming. We think that, owing to referential transparency, functional programs are usually easier to understand than imperative programs.

Algebraic Specifications. This approach composes well with functional programming. Moreover, it is conceptually close to the kind of mathematical models used increasingly within AI.

Literate Programming. Knuth's idea of programs as literature [4], intended primarily for humans to read, rather than for computers to execute, fits well with our purposes.[1]

Our interest in functional programming and algebraic specifications lead naturally to our contemplating using ML. However, we were reluctant at first to take this step. Within AI generally, and also within our institute, Common Lisp has established itself as the dominant language. Presumably, the intended audience of our publications would be familiar with Lisp, but not necessarily ML. Nonetheless, we decided that the benefits outweighed the risks in our case, and chose ML.

The toolbox now contains a sizeable library of modules. Some may be of general interest, such as those for sets and streams, and often have multiple implementations. Example AI specific modules are those for symbolic expressions, theorem proving, reason maintenance and heuristic search. Of course, there are also functors for constructing AI planners. An experimental planner we have built "beautifies" business graphics, and communicates with a graphics editor written in Common Lisp. The `qwertz` toolbox will be made freely available sometime soon.

The rest of this paper explains the reasons for our preferring ML at the time, discusses some of the problems we faced, including organizational ones, and reflects about the wisdom of our choice after three years of actual ML use. We also take this opportunity to provide some feedback to those responsible for the evolution of the language, and those developing ML programming environments.

2 Why ML?

Of all the features which persuaded us to choose ML, three stand out: compile time type checking, MacQueen's module system, and ML's small size. The principal arguments against static typing have been: 1) type declarations are awkward in an interactive language; and 2) existing type disciplines prohibit an unacceptably large class of correct programs, making it very unwieldy to implement certain abstract data types, such as lists and other "containers". Type inference and polymorphism have solved both of these problems quite well. Some Lisp programmers familiar with ML still maintain that ML's type system is too restrictive, as it does not support union types. However, we have never missed this ability in practice. Although there is at least one Common Lisp implementation which supports static type checking, ironically this requires a large number of type declarations. We prefer ML's approach.

Perhaps it is not obvious why we should be so interested in compile time type checking, if our main interest is the clear description of a software system. Does the fact that a Lisp system checks types at runtime make a correctly typed Lisp program any less comprehensible to the human reader? Presumably not; but being able to catch all type errors before publication is very comforting to human

[1] We have developed an SGML-based system for writing literate programs in any programming language [2]. The system is available for free by anonymous ftp. Contact the authors for more information.

authors. For us, a good type checker is a more important programming environment tool than a tracer or debugger. An ideal environment would have all three.

Regarding ML's module system, its main attraction is its obvious relationship to algebra. There is nothing comparable in Common Lisp. The language constructs which come closest to ML's signatures, structures and functors are Lisp packages and CLOS, the Common Lisp Object System. However, the differences are more numerous than the commonalities. Suppose one tries to compare structures to packages. There is a single, global naming environment for Lisp packages, which makes them quite inadequate for programming in the large. Suppose two programming teams inadvertently use the same package names? Without access to source code, there is no way for package users to resolve such conflicts. Even with source code, it is a thankless, tedious job. ML structure names are dereferenced at compile time, making it easy, usually, to find a loading order which avoids such problems. There are several other significant differences between structures and packages, such as the fact that structures may be organized hierarchically, but there is no space to discuss these further here.

A comparison of CLOS and ML's module system may be an interesting subject, but we can only scratch the surface here. The specification of the CLOS portion of Common Lisp alone dwarfs all of ML. For us, this is a major limitation. It would be too much to expect our poor readers to learn all of Common Lisp, including CLOS, before they can comprehend our system. Alternatively, we could have taken the trouble to define some subset suitable for our didactic purposes, but it is more convenient to be able to refer to a number of excellent, existing texts on ML, such as [5].

However, there are deeper reasons for preferring ML's module system. The object-oriented paradigm, at least as it is usually realized, unduly mixes specification and implementation levels. A class is not (only) a specification of an object, but also a generator of its instances. It serves the role of both signatures and functors in ML. Also, whereas membership of an object in a class is determined by a simple search of an inheritance graph, whether or not an ML structure satisfies a signature is determined by pattern matching, which is considerably more powerful and flexible. Finally, classes have elements of both modules and abstract data types, but are less than optimal in either role. This said, one advantage of CLOS, and other object-oriented systems, is the ease with which some abstract data types can be extended. We have a bit more to say about this below, in our wish list for ML.

There are a variety of other reasons why we preferred ML for our project, but there is only room to mention one more: ML is small, but not too small. It includes all the features we feel we need, such as exception handling and a powerful module system. Some complain about the minimal size of ML's standard library. But, for our purposes, this was more of a feature than a bug; it reduces the amount our readers must know before they can begin to understand our system.

3 Programming Guidelines

It is possible to write incomprehensible programs in any language, and ML is no exception. We have adopted a few conventions which we think make our toolbox easier to understand and use. These are only guidelines. There have been occasions to deviate from them. Only the most significant guidelines will be mentioned here.

Avoid Imperative Features. Presumably there is no need to justify this to a functional programming audience. But we are happy that ML does have imperative features. When published algorithms we needed are clearly imperative, we implemented them in the same way, rather than trying to invent functional alternatives.

Avoid Milner Polymorphism. This guideline may appear somewhat strange. However, modules provide a better way, in our opinion, to realize polymorphic types in most cases. Equality types and

the difficult system of weak type variables are two indications of problems with Milner's approach. Why should there be a special language construct for one abstract data type? We use an `EQ` signature instead, with an **`element`** type and an `eq` function, which may be any equivalence relation. To date, we have violated this guideline only once, in our **`STREAM`** signature for "lazy" sequences. How could `flatten : 'a stream stream -> 'a stream` be realised without Milner style polymorphism?

Do Not Always Use Closed Functors. Some consider it good practice to use only closed functors, i.e. to pass all substructures as functor parameters and to avoid free references to other functors. We cannot agree. The design of a functor should respect the needs of its intended users. In a functor for symbol tables, for example, another functor for hash tables may be used. Most users of symbol tables will be happy to delegate responsibility for choosing a suitable hash table implementation.

Declare Only Meaningful Types in Signatures. Types are "meaningful" only if some set of operations have been defined for them. In a signature for tables, e.g., it makes little sense to include a `key` type when this type is completely unconstrained. Instead, include some comparison operation for keys or include a `Key` substructure constrained by some appropriate signature.

Do Curry Function Definitions. This is another area where ML offers too much flexibility. If some functions are curried and others not, willy nilly, the poor programmer tends to forget which choice has been made, and the syntax of function calls tends to be an ugly mess. We chose, somewhat arbitrarily, to curry function definitions.

4 Surviving in a Lisp World

In the AI division of our research institute, the *de facto* standard language is Common Lisp. Our choice of ML has not made it easier to cooperate with our colleagues. Our original justification was that we were using ML to *specify* our system, implying we would someday *implement* the system in Common Lisp. We have been doing our best to avoid the inevitable. If you find yourself in a comparable situation, here is a tip.

To facilitate communication between Lisp and ML processes, in either direction, we have implemented an s-expression module for ML. This allows us to read and write most Lisp data structures. The concrete syntax of our planning language, for example, uses s-expressions in the traditional way. This makes it easy and familiar for others to use our planners as if they were working with a special purpose AI language embedded in Lisp. Moreover, we have written a simple CLOS class for creating and using instances of our planners. Lisp users need not know that the constructor function for the planner starts an ML subprocess. What they don't know won't hurt them.

The overhead incurred by sending s-expressions between Unix processes is negligible, at least for our application. Also, if the ML implementation has an application generator which removes all unreachable code, such as the compiler, this arrangement does not place excessive additional demands on memory. The runtime data structures we generate in ML would also have had to be generated in Lisp.

This approach is something of a crutch, to make up for the lack of some way to share functions and other values, regardless of which language they are written in. We have a few more words to say about this subject below.

5 Wish List

Let us take this opportunity to make a few suggestions to ML implementors. Some of them may not be very constructive, for which we apologize, as it is probably too late to do anything about them. The

comments are grouped into those concerning the language itself, and suggestions for future programming environments.

First the less constructive comments: We are not very happy with some aspects of ML's syntax. It is too complex, both to read and, especially, to write. The main sources of the problem seem to be the redundancy of curried and uncurried functions, as mentioned previously, and unfortunate interactions between infix operators and the left to right evaluation strategy. It is not always clear where one needs parentheses. Honestly, how many of you were not irritated that first time your ML compiler complained about a Boolean expression like `foo x andalso bar x`? If the left conjunct does not require parentheses, why should the right one?

This syntax problem brings us to another gripe. ML was originally developed because of some weaknesses of Lisp. However, incomprehensibly, several good features of Lisp did not find their way into ML, such as 1) built-in data types for symbols and symbolic expressions, for easy input and output of most structured data and, although we haven't needed this feature ourselves 2) the representation of programs as symbolic expressions, making it easier to manipulate programs as data. We have added symbols and s-expressions to our toolbox, but they would be easier to use if there were syntactic support for them, just as there are special syntactic means to construct numbers and strings.

Here is a constructive suggestion: The only feature of object-oriented programming which we sometimes miss in ML is the ability to extend data types across module boundaries. There may be numerous theoretical and practical difficulties which we do not appreciate, but perhaps it would be possible to modify ML's **datatype** construct as shown in the following example:

```
functor MyColors (C : sig
                        datatype color = red | blue
                        val paint : color -> unit
                      end) =
struct
        datatype color = C.color | yellow
        extend C.paint yellow = ...
end
```

Structures generated by this functor would have this principal signature:

```
sig
        datatype color = red | blue | yellow
        val paint : color -> unit
end
```

As **datatype** declarations are already permitted in signatures, this suggestion does not appear to violate any abstraction barriers. The new key word, **extend**, is used instead of **fun** to continue the definition of a function to handle the new cases.

In object-oriented languages, an object is an instance not only of its parent, but also of all ancestors of its parent. Analogously, it would be useful if all structures matching the extended signature also matched the "parent" signature. To achieve this effect, the interpretation of data types would have to be modified to mean that the type includes *at least*, rather than exactly, the constructors listed. Failure to handle some constructor not mentioned in the signature, but part of the actual structure used, should presumably raise a runtime **Match** error.

Here is our final language suggestion. Using **include** it is possible to extend signatures. A comparable **exclude** declaration, for subtractive "inheritance", would be useful for minimizing dependencies in functor definitions.

Let us now turn to environmental issues. Our comments here are based on our experiences with SML of New Jersey, which we can highly recommend. But nothing is perfect.

First, SML of NJ requires enormous amounts of RAM. We fail to understand why ML should be any more memory hungry than, say, Macintosh Common Lisp, which is quite useable with as little as 2 MB of RAM. One reason for this problem may be that all modules used in a program need to be loaded at once, during development. When an application is "exported", memory demands can be radically reduced, due to dead code analysis and, of course, because the compiler is no longer necessary. The compiler may also be used in batch mode, but in this case it produces binary files in another, incompatible format, and does not perform type-checking across module boundaries. Our first wish would be an improved batch compiler without these shortcomings. To test programs, the required binaries could be loaded into a minimal environment allowing expressions to be evaluated, but no new functions or modules to be compiled. An interpreter should suffice. It would also be nice if there were some way to explicitly unload modules, to avoid having to restart ML when memory starts getting tight.

Finally, it would be helpful if an ML compiler would produce object code compatible with others languages. Ideally, the object code format would be the same as that produced by Unix C compilers. This may be the single most important improvement, if one is interested in increasing ML's popularity. Imagine gaining access to POSIX or X Window libraries just by defining appropriate ML signatures. However, if this is not feasible, it would be quite useful if at least the functional programming community, including Lisp and Scheme, would adopt some standard object code format suitable for this class of languages. Poplog, which supports mixed ML, Common Lisp, Prolog and Pop-11 programming, demonstrates this can be done, but it is not a standard.

6 Would We Do It Again?

Yes, definitely. Despite our complaints, we still are convinced that we made the best choice. No better alternative has appeared within the last three years. Nor are there signs of anything better on the horizon.

References

[1] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning.* Morgan Kaufmann, San Mateo, CA, 1990.

[2] Thomas F. Gordon. The `qwertz` SGML Document Types, Version 1.1 Reference Manual. Working Paper 588, German Research Center for Computer Science (GMD), November 1991.

[3] J. Hertzberg and A. Horz. Towards a Theory of Conflict Detection and Resolution in Nonlinear Plans. In *Proc. IJCAI-89*, pages 937–942, 1989.

[4] D.E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[5] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[6] W. Swartout. DARPA Santa Cruz Workshop on Planning. *AI Magazine*, 9(2):115–131, 1988.

[7] S. Thiébaux and J. Hertzberg. A Semi-Reactive Planner Based on a Possible Models Action Formalization. In J. Hendler, editor, *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS92)*, San Mateo, CA, 1992. Morgan Kaufmann. To appear.

Attribute Grammars in ML

Sofoklis G. Efremidis*
Cornell University
sofoklis@cs.cornell.edu

Khalid A. Mughal†
University of Bergen‡
khalid@cs.cornell.edu

John H. Reppy
AT&T Bell Laboratories
jhr@research.att.com

April 10, 1992

1 Introduction

Attribute grammars provide a formalism for assigning meaning to parse trees of a context-free language [Knu68]. Because of their syntax-directed form and declarative style, they provide a useful notation for specifying compilers [KHZ82] and language-based editors [RT88]. This paper reports on a system we are developing, called AML (for *Attribution in ML*), which is an attribute grammar toolkit for building language-based editors using SML [MTH90].

AML is a spiritual heir to the Synthesizer Generator project at Cornell University [RT88], which focused on efficient incremental evaluation techniques, and the Pegasus project at AT&T Bell Laboratories [RG86], which focused on providing a high-level foundation for interactive systems. In our system we plan to build on the evaluation technology of the Synthesizer Generator while using a higher-level foundation for the implementation.

In the next section, we give some background about attribute grammars. Then we describe our specification language for attribute grammars, followed by a description of the internals of our system. Lastly, we discuss the project's status and future plans.

2 Attribute grammars

An attribute grammar is a context-free language (CFL) together with a set of attributes for each nonterminal and a set of attribute evaluation rules for each production. An attribute is either *synthesized* or *inherited*; for each production $p : X_0 ::= X_1 \ldots X_{n_p}$, there are evaluation rules that define the synthesized attributes of X_0 and the inherited attributes of $X_1, \ldots, X_{n_p}$.

2.1 Attribute evaluation

Each node of a parse tree in the underlying CFL is labeled with *instances* of the attributes associated with the nonterminal at the node. The evaluation rules define a set of constraints on the attribute instances, and computing the attribute values can be viewed as a constraint solving problem. An attributed tree is said to be *consistent* if its attribute values satisfy the constraints defined by the attribution rules. The simplest way to solve this problem is to topologically sort the attribute instances by their dependencies and evaluate in topological order. For many grammars, however, more efficient and specialized evaluation strategies can be used. Attribute grammars are classified by their evaluation strategies; for example, the parser generator yacc implements a grammar in which all attributes are synthesized and can be evaluated in a single bottom-up pass. One important class of attribute grammars is the class of *Ordered Attribute Grammars* (OAGs) [Kas80]. OAGs

*This work was supported, in part, by NSF Grant ASC–8800465 and ONR Grant N00014–91–J–4123.
†Support for this work provided, in part, by the Norwegian Research Council for the Sciences and the Humanities, and the University of Bergen.
‡Author's current address: Cornell University, Ithaca, NY.

are interesting because they include many useful "real-world" grammars, and because they can be efficiently evaluated by fixed-plan evaluators.

2.2 Subtree replacement and incremental evaluation

In his seminal thesis, Reps showed that attribute grammars provide a useful formalism for defining language-based editors [Rep82]. Such systems use attributed abstract syntax trees to represent programs being edited and map editing operations to subtree replacements (e.g., a delete operation is implemented by replacement with a null tree). After a subtree replacement operation, the tree will no longer be consistent; Reps and others have described so called "optimal" evaluation algorithms for restoring attribute consistency [Rep82, Yeh83, Hoo87, Hud91]. Applications of this technology include structured editors for programming languages, interactive theorem provers [Gri87] and code generators [Mug88].

3 Attribution in ML

Our approach to supporting attribute grammars in ML is a fairly conventional generator-based approach. Our system takes an attribute grammar specification and generates a tree machine for building and manipulating abstract syntax trees and an evaluator for the grammar. These are combined with grammar-independent modules to construct a complete system. We chose this approach over embedding the system in the interactive ML system, because it allows more flexibility for analysis and optimization. The remainder of this section describes our specification language.

3.1 The specification language

The AML specification language has two parts: a *core* language for specifying grammars and a *module* language for structuring large specifications into separate units. The core language is sufficient to specify an attribute grammar as a monolithic specification, but it is often useful to structure large specifications in a modular fashion.

An attribute grammar specification in AML consists of three main parts: the abstract syntax, the associated attributes, and the attribution rules. Chapter 4 of [RT88] gives a sample specification of a language-based editor for a simple imperative programming language; we use this example to illustrate our specification language.

The abstract syntax is declared using a limited form of the ML datatype declaration, which we call a *prodtype* declaration. For example, the abstract syntax of expressions can be declared as follows:

```
prodtype exp
  = EmptyExp
  | IntConst of int
  | True
  | False
  | Id of string
  | Equal of (exp * exp)
  | NotEqual of (exp * exp)
  | Add of (exp * exp)
```

This defines a nonterminal type exp with a set of productions. Prodtype declarations are more restrictive than datatype declarations in that the argument types of the productions (constructors) are required to be tuples of ground and nonterminal types.

The attributes associated with the nonterminals are declared separately from the abstract syntax. For example, the following declaration states that expressions have a synthesized type attribute and an inherited typing environment:

```
datatype type_t = BoolTy | IntTy | EmptyTy

attribute exp with
     synth typ : type_t
     inher env : (string * type_t) list
   end
```

The last part of the specification are the rules for attribution. These require notation for naming the nonterminals of a production and the attributes of a nonterminal. We use ML-style pattern matching, with minor extensions, to bind names to the nonterminals and use the notation $X\$a$ to refer to attribute a of X. The following are the typechecking rules for expressions in the simple editor:

```
attribution e0 : exp
  = EmptyExp                          with rule e0$typ = EmptyTy end
  | (IntConst _)                      with rule e0$typ = IntTy end
  | (True | False)                    with rule e0$typ = BoolTy end
  | (Id id)                           with rule e0$typ = lookupType(id, e0$env) end
  | (Equal(_, _) | NotEqual(_, _)) with rule e0$typ = BoolTy end
  | (Add _)                           with rule e0$typ = IntTy end
  | (Equal(e1, e2) | NotEqual(e1, e2) | Add(e1, e2)) with
       val error = if (incompatibleTypes(e1$typ, e2$typ))
             then NONE else SOME " TYPE ERROR "
       rule e1$env = e0$env
       rule e2$env = e0$env
     end
```

In these rules, e0 is bound to the left hand side of the productions. Each clause defines a production pattern with a collection of rules, which are ML expressions. We use the *or-pattern* notation to group several productions that have identical rules (e.g., the last clause). The rules for a given production can also be factored across several clauses (e.g., the rules for `Equal`). The `val` binding in the last clause defines the *local attribute* `error` for those productions. Local attributes are associated with productions, and are typically used to factor common subexpressions and, as in this case, to communicate predicates about the node's state to external observers. For example, a pretty printer might highlight a node's subtree when the `error` attribute is not `NONE`.

3.2 Modular specifications

Attribute grammar specifications tend to be considerable in size and non-trivial in complexity. Attribute grammars as such do not provide any means for modularization. As pointed out in [Kas91], an effective strategy is to allow a module for an AG to contain the attribution of one *semantic* aspect only. Several approaches have been advocated to modularize AGs [DC88, FMY92, Far92]. While these approaches address the problems of partitioning a grammar specification, they do not provide the concept of a *module*.

Our approach is based on the module facility in ML. *Grammar structures*, akin to ML structures, allow encapsulation and *grammar signatures*, akin to ML signatures, provide information hiding (i.e., the interface to the outside world). A grammar structure specifies a particular semantic aspect of the problem being expressed as an attribute grammar. A grammar structure can have several grammar signatures providing different views of the module. A relevant grammar signature is used by another module to import exactly the information it requires. Grammar signatures allow information about productions of the grammar and the attributes of the grammar symbols to be imported and exported in a controlled manner, aiding comprehensibility and maintainability.

3.3 Computation with attributed terms

Since `prodtype` declarations in AML are just a restricted form of `datatype` declarations in ML, it is straightforward to express computations on the unattributed view of a tree. This is especially useful

for supporting features such as *syntactic reference* (i.e., using a subtree as an attribute value) and *higher-order attribute grammars* [VSK89, TC90].

It is also desirable to allow user computations on attributed terms. To support this, we generate an abstract interface to the tree machine that provides for safe manipulation of attributed terms. Operations include attributing unattributed terms, reading attribute values and tree editing (with reattribution). This framework allows for a general and uniform platform for building applications that can manipulate attributed terms and allow access to attribute values

4 Implementing attribution in ML

A number of issues must be addressed in the implementation of the tree editor and evaluator:

- Navigation and subtree replacement operations for attributed trees must be supported.
- It should be possible to convert between attributed and non-attributed versions of terms. Attribution of a non-attributed term is done by the evaluator; the converse should also be possible.
- A mechanism for associating attributes with tree nodes must be provided. It is especially useful for this mechanism to support sparse attribution (e.g., because of copy rules). In addition, operations for accessing and setting the values of attributes must be provided.

The Synthesizer Generator uses a heavy-weight tree-node representation that relies on mutable fields in the tree nodes (attribute instances, parent and child pointers, and other status fields). This representation supports efficient navigation, tree editing and attribute evaluation, but does not support sparse attribution, sharing of trees and easy mapping between values computed by user code and abstract syntax trees. Furthermore, the heavy reliance on mutable fields does not map well to ML, since it requires `ref` cells, which add extra space and garbage collection overhead.

Our approach is light-weight: we use the `datatype` equivalent of the `prodtype` declarations as our tree representation, and use *paths* to label nodes and to support navigation. Mutable state, such as attribute values, is held in auxiliary data structures that map tree nodes to their state. In this scheme, an attributed tree is just a pair of an unattributed tree and a map from nodes in the tree to their attribute instances.

4.1 Abstract syntax trees and paths

The `prodtype` declarations in an AML specification are translated to the equivalent ML `datatype` declarations. Because it is often necessary to refer to an arbitrary tree node, we generate type `tree_t`, which is the discriminated union of the nonterminal and terminal types. We also generate functions for examining, visiting and replacing the children of a tree node. These types and functions are contained in a structure that matches the following grammar-independent signature:

```
signature TREE =
  sig
    type tree_t
    exception Child
    val isLeaf : tree_t -> bool
    val nChildren : tree_t -> int
    val children : tree_t -> tree_t list
    val nthChild : (tree_t * int) -> tree_t
    val replaceNth : (tree_t * int * tree_t) -> tree_t
  end (* TREE *)
```

The generated structure has a richer signature that is used by other grammar-dependent parts of the system.

To specify a node in a tree we use a path from the node to the tree's root. Paths and their operations are defined in a grammar-independent functor, called `PathFUN`, which is parameterized by a structure `T` matching the `TREE` signature. The path type is:

```
datatype path_t
  = Root of T.tree_t
  | Path of (int * T.tree_t * path_t)
```

The path `Path(i,t,p)` specifies node t, where t is its parent's i'th child and p is the path to t's parent. One of the advantages of this scheme is that it promotes sharing of trees (i.e., *dags*), since a single physical node can have several different paths. The functor `PathFUN` provides functions for examining the children of a tree node that is specified by a path, tree navigation and subtree replacement.

4.2 Attribution

We have chosen to store attributes and other mutable state information in auxiliary data structures. These data structures map tree nodes, specified by paths, to state information. There are a number of possible representations of these auxiliary maps, we are currently using hash tables keyed by hashing a node's path. An alternative representation that we are considering is a shadow tree.

By storing attribute instances in auxiliary structures, we gain several advantages. First, it means that the existence of an attribute instance for a particular node is completely independent of the node's representation. This allows us to trade time for space, by recomputing attribute instances instead of caching them in the tree. It also means that we can go back and forth between attributed and non-attributed trees without rewriting (although attributing a tree does require attribute evaluation).

There are two possible interfaces to the auxiliary data structures that hold the attribute instances of a tree. In the generic (or grammar-independent) approach, there is a single type of all attribute values and two functions to get and set an attribute value of a given tree node. The principal advantage of this approach is that it allows the implementation of the auxiliary state to be grammar-independent. In this approach, the code for the attribution rules must inject and project values from the attribute type. The grammar-dependent approach is to generate specalized functions for getting and setting each nonterminal's attributes. This approach is more efficient, since it exploits compile-time information to avoid run-time type discrimination. In our current prototype evaluators, which are hand-coded, we are using the former approach.

5 Status and Future Work

We have defined the core of our specification language and sketched the AML module facility. We have been prototyping various parts of the run-time support system (such as the tree machine and paths). We have also implemented some hand-written attribute evaluators for small grammars. We are now implementing a compiler for our specification language.

There are a number of issues that we are planning to examine in the future:

- Address issues of input and output: pretty printing of trees and editing. We plan to build this on top of eXene [GR91].
- Support a modular form of attribute grammar specifications as stated previously.
- Our use of ML-style pattern matching in the attribution rules gives us something very close to *attribute patterns* [Far92]; we would like to explore this similarity.
- Allow the use of type constructors, such as `list`, in the declaration of prodtypes. One can imagine generalizing this to arbitrary parameterized prodtypes.
- Investigations of incremental attribute evaluation schemes with a focus on space/time tradeoffs and parallel evaluation [KG92].
- The implementation of real applications using AML; in particular, a programming environment for SML.

A forthcoming technical report [EMR] provides more details and discusses many of the issues raised in this abstract.

6 Acknowledgements

We would like to thank Tim Teitelbaum, David Gries, Sanjiva Prasad, Chet Murthy and Aswin van den Berg for stimulating discussions pertaining to this project.

References

[DC88] Dueck, G. D. and G. V. Cormack. Modular attribute grammars. *Technical Report CS-88-19*, Faculty of Mathematics, University of Waterloo, Canada., May 1988.

[EMR] Efremidis, S. G., K. A. Mughal, and J. H. Reppy. AML: Attribute grammars in ML. Technical Report, Department of Computer Science, Cornell University (*in preparation*).

[Far92] Farnum, C. Pattern-based tree attribution. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, January 1992, pp. 211–222.

[FMY92] Farrow, R., T. J. Marlow, and D. M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, January 1992, pp. 223–234.

[GR91] Gansner, E. R. and J. H. Reppy. eXene. In *Third International Workshop on Standard ML*. Carnegie Mellon University, September 1991.

[Gri87] Griffin, T. G. An environment for formal systems. *Technical Report 87-846*, Department of Computer Science, Cornell University, June 1987.

[Hoo87] Hoover, R. *Incremental Graph Evaluation*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York 14853, May 1987. Also Tech Report 87-836.

[Hud91] Hudson, S. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, **13**(3), 1991, pp. 315–341.

[Kas80] Kastens, U. Ordered attribute grammars. *ACTA Informatica*, **13**(3), 1980, pp. 229–256.

[Kas91] Kastens, U. Attribute grammars as a specification method. In *Attribute Grammars, Applications and Systems*, pp. 16–47. Springer-Verlag, 1991. Lecture Notes in Computer Science 545.

[KG92] Kliber, A. and M. Gokhale. Parallel evaluation of attribute grammars. *Transactions on Parallel and Distributed Systems*, **3**(2), March 1992, pp. 206–220.

[KHZ82] Kastens, U., B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, vol. 141 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[Knu68] Knuth, D. E. Semantics of context-free languages. *Mathematical Systems Theory*, **2**, 1968, pp. 127–145.

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass, 1990.

[Mug88] Mughal, K. A. *Generation of Runtime Facilities for Program Editors*. Ph.D. dissertation, University of Bergen, Norway, 1988.

[Rep82] Reps, T. W. *Generating Language-Based Environments.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, NY, 1982. Published by The MIT Press, 1984.

[RG86] Reppy, J. H. and E. R. Gansner. A foundation for programming environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986, pp. 218–227.

[RT88] Reps, T. and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-based Editors.* Springer-Verlag, New York, NY, 1988.

[TC90] Teitelbaum, T. and R. Chapman. Higher-order attribute grammars and editing environments. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, pp. 197–208.

[VSK89] Vogt, H., S. Swierstra, and M. Kuiper. Higher order attribute grammars. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1989, pp. 131–145.

[Yeh83] Yeh, D. On incremental evaluation of ordered attributed grammars. *BIT*, **23**, 1983, pp. 308–320.

ML and Parsing—a Position Paper

Nick Haines
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

April 13, 1992

Abstract

I would like to bring together several issues connecting ML and parsing. First, the parsing of Standard ML is not simple, as the grammar is not LR(k), so the parser in an ML system must be more sophisticated than for some other modern languages. Second, ML presents an opportunity to escape from the traditional type-unsafe parser generation technique of source-code manipulation. Third, there is considerable work being done on extensible syntax in other languages (and some on quotation and anti-quotation in ML), and there is a natural and clean technique, using parsing technology, allowing the user to extend the ML syntax arbitrarily.

1 Parsing ML

The ML grammar[4] is not suited to simple parsing. The definition was written before any parser was constructed for the language, and was not inspected for any of the common desirable grammar properties (such as LALR(1)). It is not LALR(1) in several places, and not LR(k) in at least one (typed sub-patterns). Infix and precedence declarations make parsing of application expressions so awkward that this is generally handled in a post-parse phase. None of the current range of ML systems accept the full language.

There should certainly be work within the ML community to decide a new standard grammar for ML, and there has been some progress towards this. However, while we have this definition, it seems wise to work as closely to it as possible in all regards, and to build a parser that accepts the full language as defined. So we need a parsing algorithm which will accept the ML grammar. There are several such techniques [2, 7], and some recent work has been done by Jan Rekers on parser generation for these parsing algorithms[5]. However, none of these techniques have yet been applied to ML. Rekers' algorithm is particularly useful as it takes standard LR(k) parsing tables (usually k =0 or 1, for small tables) and is very efficient for grammars which are 'nearly' LR(1).

2 Parser Generation in ML

Traditional parser generators[3] are text manipulation programs, which read files written in some grammar description language, calculate (typically LALR(1)) parsing tables for the grammars, and cut-and-paste sections of 'source code' (unparsed strings) around a textual representation of the parsing tables in order to produce the source of a parser, output as pure text. This is an unsatisfactory approach, with several problems:

- one needs to define a language for describing grammars, and read (i.e. lex, parse, and error-check) raw text files in this language. The language has no computational power, and is simply a way of providing the parser generator with the hierarchical structure of a grammar and a set of associated strings for the actions. It is not even possible to provide expressions for these strings, to be evaluated at generation time, except in a very restricted sense (unless one chooses to add another layer of generation and write code to produce the original text file).
- one must write 'disembodied' sections of code, which are not compiled until after the parser generation. Compiler error messages which emerge at this stage must be traced by examining the generated source and extrapolating back to the original source error. Making corrections then requires the parser generation to be re-run.
- the output parsing code is bound together with the parse tables in a single object. One cannot change the parse tables and retain the parser without recompilation. So allowing extensible syntax is difficult.
- the 'make' actions for the grammar are completely different to those for the other parts of the language system. If one is using a specialised make system (for instance, one acting solely upon ML source files), the parser generation must be dealt with separately.

(Note: there are some sophisticated systems, such as those used at INRIA, which go some way to ameliorate the first two problems, by (for instance) providing good cross-referencing of compile-time errors and a more powerful grammar-description language allowing one to specify better error-recovery behaviour. But the basic problem is caused by dealing with source text without semantics, and that is not corrected in those systems).

ML provides an alternative approach—to define the signature of a grammar (with actions), and to write a parser generator as a functor, taking a grammar as argument and returning a parser. Then the full power of ML is available for building grammars and their actions, the type-safety of ML ensures that the generated parser does not cause runtime failures (the equivalent of the compiler errors in the previous model), a suitable structure allows different parse tables to

be exchanged for extensible syntax, and the grammar can be simply integrated into the ML separate compilation world.

Various work has been done on this in the past, particularly by Nick Rothwell at LFCS[6]. I have worked on this code, and hope to produce a more fully integrated version.

One reason why this approach has not been commonly employed before is that there are type difficulties, seemingly requiring the different non-terminals of the grammar to have a common type. The typical solution is to define a single sum type (using the **datatype** declaration) of all the node types of the abstract syntax. This introduces a layer of constructing and deconstructing (with exception-raising on failed matches) without adding any real type-safety, and the cost of these additional functions is high, since they must be called on every 'reduce' action of the parser. I have been working on a new approach, using simple pairs of functions which should be eliminated by a good optimizing compiler.

Such a functor approach can also be applied to lexer generation (and has been so applied), and I shall be integrating parser and lexer generation in this way (so that they share a 'token' type, and present a single interface to the rest of a compiler). Both parser and lexer generators are general, and can be applied to other languages than ML. The parsers generated use Rekers' algorithm, so can be applied to any context-free grammar.

3 Extending the ML syntax

The New Jersey compiler[1] is largely written in ML, and has lexers and parsers generated using traditional techniques. These can be replaced by functorized versions. Given a clear, public definition of the abstract syntax used by the parser, and public access to the parser generator, users can define their own concrete syntax for ML. By separating the parse tables (generated by the parser generator) from the parser itself, the user can be allowed to 'switch in' their own parse tables, including parts of their own language in their ML code, thereby extending the syntax. Specialized syntaxes can be defined for particular problems, and user code and data can be written in their own form in a type-safe, controlled way.

I have been working on the New Jersey parser with this goal, and although there are some difficulties (the parser is quite tightly integrated to the rest of the compiler, especially for error-handling), they are not insuperable.

Ultimately, this tool provides a technique for the concrete syntax of ML to be 'corrected' without losing the abstract syntax on which the semantics, which are commonly perceived as the main strength of the language, are based.

References

[1] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987. Volume 274 of *Lecture Notes in Computer Science*.

[2] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[3] S. C. Johnson. Yacc—yet another compiler-compiler. Computing Science Technical Report 32, AT & T Bell Laboratories, 1975.

[4] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[5] Jan Rekers. Generalized LR parsing for general context-free grammars. Technical report, CWI, 1991.

[6] N. Rothwell. source code and examples of functor-generated parsing. personal communication to the author.

[7] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.